DTIC FILE COPY

(2)

WORKING MATERIAL

for the Lectures of

Robert L. Constable

Implementing Metamathematics as an Approach to Automatic Theorem Proving

AD-A213 960

DTIC
ELECTE
OCT 31 1989
S D
B

INTERNATIONAL SUMMER SCHOOL

ON

LOGIC, ALGEBRA AND COMPUTATION

MARKTOBERDORF, GERMANY, JULY 25 - AUGUST 6, 1989

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

89 10 27 026

THIS SUMMER SCHOOL IS ORGANIZED UNDER THE AUSPICES OF THE TECHNISCHE UNIVERSITÄT MÜNCHEN AND IS SPONSORED BY THE NATO SCIENCE COMMITEE AS PART OF THE 1989 ADVANCED STUDY INSTITUTES PROGRAMME. PARTIAL SUPPORT FOR THE CONFERENCE WAS PROVIDED BY THE EUROPEAN RESEARCH OFFICE AND THE NATIONAL SCIENCE FOUNDATION AND BY VARIOUS INDUSTRIAL COMPANIES.

Implementing Metamathematics as an Approach to Automatic Theorem Proving*

Robert L. Constable *Douglas J. Howe*†

Abstract

A simple but important algorithm used to support automated reasoning is called *matching*: given two terms it produces a substitution, if one exists, that maps the first term to the second. In this paper the matching algorithm is used to illustrate the approach to automating reasoning suggested in the title. In Section 3 the algorithm is derived and verified in the Nuprl proof development system following exactly an informal presentation of it in Section 2.

The example serves to introduce a particular automated reasoning system, Nuprl, as well as the idea of deriving programs from constructive proofs. The treatment of this example also suggests how these systems can be soundly extended by the addition of constructive metatheorems about themselves to their libraries of results.

(KR) ←

1 Introduction

People use computers to do all sorts of things, from the mundane to the miraculous. Among the enduring uncommon uses are game playing, notably chess, and *theorem proving*. Clearly chess is interesting, but why has theorem proving remained active? We can make an argument that theorem proving is a significant activity, and if computers can help when it really counts, society will be better off. Some people are interested in it for exactly those reasons. They apply automated theorem proving in such areas as *program verification* and *hardware verification*. There are now companies whose employees make a living from this activity. But these people are mostly newcomers to the subject (see [21] for a recent survey). We might call them *logic engineers*.

There are other motives for studying theorem proving by computer. Indeed, efforts to employ computers in the enterprise have from the beginning brought into proximity researchers of diverse interests. There are those interested in studying intelligence, especially reasoning. They argue that reasoning and problem solving are critical to intelligence and that proving theorems is intelligent behavior. People with those interests will usually associate themselves with the study of artificial intelligence. Prominent pioneers in their ranks are the likes of Allen Newell, Herbert Simon, and John McCarthy. On the other hand, the subject has attracted applied logicians who study proof systems and algorithms

*This research was supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409

†Authors' address: Department of Computer Science, Cornell University, Ithaca NY 14853

for finding proofs. These people want to test their algorithms; their goal is to prove a lot of theorems fast, regardless of how. They are not necessarily concerned with emulating human thought processes unless those processes are effective in proving theorems. This early dichotomy among people "in the area" is well-known and amply referenced [9]. The contrast in the early days of the subject was between A.I. programs such as Logic Theorist [26] and fast decision procedures like Wang's [32]. Besides Hao Wang, there are people like J.A. Robinson and Martin Davis in the latter group.

Nowadays another group has become involved. They see the possibility of building a variety of tools which are especially suited to helping people use computers to prove theorems. The important point for them is that theorem proving is a complex information processing task and the computer offers new ways to perform it. The goal is to allow anyone, logician or A.I. researcher, to easily test ideas for proving theorems. This is a newer concern. The authors started in this area along with Joseph Bates and others at Cornell who will be referenced in the course of this article. One of the pioneers here was Robin Milner and his LCF group [14,27].

In short, let us say that since 1950 it has been a goal of A.I. to find architectures for intelligence. Systems like Logic Theorist, GPS and now Soar [29] are built to achieve this goal. Since the late 1950's there has been the goal to "prove lots of theorems". Systems like the Argonne prover [33], the Boyer & Moore prover [4] and others are oriented this way. Since the 1970's there has been the goal to apply the technology, as in Gypsy [13], and to build *systems* that allow easy expression of a wide variety of strategies (including in the limit those from A.I.), as in LCF and Nuprl [8].

A great deal of practical work has been put into meeting these basic goals. Now there is another goal, made possible by the substantial body of experimental results; it is to *understand* the experimental results, to develop a *theory of theorem proving.* In fact, one of the deepest results in computing theory arose precisely in pursuit of this aim. S.A. Cook proved that the tautology problem is essentially the same as the problem of membership in languages accepted by nondeterministic polynomial time Turing machines. This led Cook to propose the now famous $P = NP$ problem which lies at the heart of theoretical computer science.

Because so much is known about logic and its algorithmic content, there are many compelling questions about the automation of theorem proving—questions about why an algorithm works on one class of problems but not on another, questions about those characteristics of a sentence or its partial proof which tell an experienced prover what to do next, questions about the cost of proving a result relative to a library of results, questions about the limits to feasible automation. There is the hope that the answers to these questions may provide answers to deeper questions about the requirements for an intelligent system.

In this article we will describe the enterprise of automated theorem proving in a context that suggests an outline of a *theory of theorem proving* and suggests an organization of the various existing methods that facilitates the computer scientist's goal of providing a general environment for the empirical study of the subject. The basic idea is that automated theorem proving can be seen as the implementation of a constructive *metamathematics.* In metamathematics one proves theorems about doing mathematics. Some of these theorems have interesting computational content, such as a result claiming that a theory is decidable

or a result saying that any proof of a theorem of one form can be converted to one of another form (say from prenex form into nonprenex form). We will show how some theorem proving algorithms can be understood as implementations of theorems about proofs and formulas. Especially noteworthy are results of the second author on term rewriting [17].

One of the critical reasons for adopting this point of view is that much of the knowledge we discover about theorem proving is potentially useful to the automated system itself. In order to be used by machines, the knowledge must be formalized. Conversely, the more we can say about the algorithms we use, the more effectively we can use them. This suggests that we want to develop them in the context of a formal theory. Related reasons for doing this are discussed by Davis and Schwartz [10], Boyer and Moore [5] and Shankar [30].

One of the major advantages of formalizing knowledge about theorem proving is that we can use it to extend the stock of derived rules of inference. This in some cases allows us to avoid running theorem-proving algorithms. This point will be mentioned again in Section 3.3. To the extent that results from other parts of mathematics are useful in theorem proving, as in the use of graph algorithms in congruence closure, we want to be able to prove that they are correct and preserve the correctness of the logic when they are used in derived inference rules.

The rest of the paper is organized as follows. First we develop some informal metatheory, focusing on two particular metatheorems. One of these yields an algorithm for first-order matching. The computational content of the other is a tableau decision procedure for propositional formulas. We then present a formalization of the development of the match algorithm that was carried out in the Nuprl proof development system. This formalization is based directly on the informal account.

2 Informal Constructive Metamathematics

2.1 Preliminaries

To begin we establish some basic definitions and notations. The type of integers is denoted *int* and its elements are $0, +1, -1, \ldots$. When dealing with syntactic matters it is convenient to have a type *atom* of atomic symbols or (or "atoms"). Its elements are denoted "*id*" where *id* is a character string. We assume that an equality relation is provided on this type; we write it as $a = b$ *in atom*. We also assume that equality is *decidable*. This is expressed by saying that for all atoms a and b, either $a = b$ *in atom* or not. In general we treat the word *or* (and its symbolic form $\vee$) *constructively*; that is, when we say $P \vee Q$ for P and Q statements, we mean that either P or Q is true and we know which one. The classical meaning of *or*, as in the phrase "P *or* Q *classically*", can be taken as an abbreviation of *not* (*not* P *and not* Q). We sometimes write *not* as $\neg$, so that $\neg P$ is *not* P. We will often abbreviate *and* by $\&$ and "for all x of type A" by $\forall x : A$. So "for all x of type A, $P(x)$ or $Q(x)$ classically" is written as

$$\forall x{:}A.\ \neg(\neg P(x) \vee \neg Q(x)).$$

We will always specify the classical *or* explicitly when we need it; otherwise *or* has its constructive meaning which, as we have just seen, is convenient for expressing the notion

that a relation is *decidable*.[1]

Given two types A and B, their *cartesian product* is denoted $A\#B$ and consists of pairs $\langle a, b\rangle$ where a is in A and b is in B. Thus *int#atom* contains elements like $\langle 0, \text{"0"}\rangle$.

The type $A \to B$ denotes the *computable functions* from A to B. These are denoted as $\lambda x.exp$ for *exp* an expression denoting an object of type B. If f is a function in $A \to B$ and a belongs to A, then $f(a)$ is the *application* of f to a and is a value in B. If we want to talk about noncomputable functions, then we treat them as single-valued relations over $A\#B$. That is, a relation F contained in $A\#B$ is a *classical function* if and only if for all $\langle a, b\rangle$ and $\langle a', b'\rangle$ in F, if $a = a'$ *in* A, then $b = b'$ *in* B.

The *domain* of a classical function F is defined as the set of all a in A for which we can find an element b in B such that $\langle a, b\rangle$ is in F. Symbolically we write the phrase "we can find an element b in B such that $R(x)$" as $\exists\, x{:}\, B.\ R(x)$. We denote the set of all elements x of A such that $R(x)$ holds as $\{\, x{:}\, A \mid R(x)\,\}$. So the domain of a classical function F is

$$\{\, x{:}\, A \mid \exists\, y{:}\, B.\ \langle x, y\rangle \ \textit{in}\ F\,\}.$$

Another familiar (but little-used in this paper) classical concept is the *classical domain* of F. This consists of those elements of A for which it is not impossible to find a corresponding element of B, that is, for which a range element *classically exists*. To say that there exists classically an element y of type B such that $R(y)$ is to say $\neg(\forall\, y{:}\, B.\ \neg R(y))$. We denote this as $E\, y{:}\, B.\ R(y)$. So the classical domain of F is

$$\{\, x{:}\, A \mid E\, y{:}\, B.\ \langle x, y\rangle \ \textit{in}\ F\,\}.$$

Another important way to build types from others is to unite them. Given types A and B we write their *disjoint union* as $A|B$. It is essential that we can tell for any element of $A|B$ whether it is from A or B. One way to do this is to write the elements of the union so that we can tell. We take the elements to be $inl(a)$ and $inr(b)$ for a in A and b in B. The operator *inl* stands for *inject left* while $inr(b)$ stands for *inject right*. So even when we form the type $A|A$, we can tell for any element of this union whether it is a "left a" or a "right a". In set theory the disjoint union is usually defined from the ordinary union by providing a scheme for tagging elements (see [3]).

For any type A, another important type built from it are lists of elements from A. The type is denoted A *list*. All nonempty lists are built from the empty list which is denoted *nil* (regardless of the type A). If a is an element of A, then $a.nil$ is a nonempty list. The dot operation is sometimes called *cons* for *construction*. In general given a list t and an element h of A, $h.t$ is a list with h as its *head* and t as its *tail*. It is convenient to write the list $a_1.a_2.\cdots.a_n.nil$ as $(a_1, \ldots, a_n)$. We sometimes do this in the informal mathematics. Below we will see how the type A *list* is a special case of the more general notion of an inductively defined type.

One important kind of construction arises from the inductive nature of lists. We can build an object incrementally from the elements of a list following the same pattern used to build the list. That is, a list starts from the empty list *nil*. Suppose that corresponding

[1] In the case of the integers, we can either take equality as a primitive decidable relation, $x = y$ *in int*, or we could define it inductively. We take it as primitive. In order to take these equality relations as primitive, there must be an algorithm for deciding them. In the case of integers and atoms this is obvious.

to it we can specify some object, such as the integer 0. Given a list t, we extend it by adjoining an element to form $h.t$. In the same way, given a value associated with t, say its length v, we can build a value associated with $h.t$ incrementally, say by adding one to v, thereby associating $v+1$ to $h.t$. This process of associating a value with a list is sometimes called *primitive recursion on lists* or *list recursion* or *list induction*. One way of writing it is illustrated by the following definition of the length of a list.

$$\begin{aligned} len(nil) &= 0 \\ len(h.t) &= 1 + len(t) \end{aligned}$$

The critical components in this definition are an expression b for the value in the base case, names for the components of the arbitrary list, say h for its head and t for its tail, a name v for the value associated with t, and finally an expression e telling how to build a new value given h, t and v. In the example, e is $1 + v$; generally it is an expression in h, t and v. Instead of writing such recursion equations we use a linear format to present the information:

$$list\_ind(l; b; h, t, v.e)$$

(*list_ind* for "list induction"). In this format, l is the list expression; b is an expression defining an object, say of type T, in the base case, and e is an expression which defines an object, of type T, in the induction case. The expression e can refer to the variables h, t and v. These variable occurrences of h, t and v before e are *binding occurrences*, and they *bind* in e. The definition of length in this format is

$$list\_ind(l; 0;\ h, t, v.\ v + 1).$$

This *list_ind* form specifies a computation when we give the explicit rules for evaluating it. To write these rules we need an informal notation for substitution. Given an expression t with a free variable x, and a term s, then $t[s/x]$ denotes the expression t with s substituted for each occurrence of x in t. More generally, if $x_1, \ldots, x_n$ are variables and $s_1, \ldots, s_n$ are terms, then $t[s_1/x_1,\ s_2/x_2, \ldots, s_n/x_n]$ *represents* the simultaneous substitution of each s_i for x_i. A formal definition of substitution is given later. With this notation the rules are:

$$\begin{aligned} list\_ind(nil; b; h, t, v.e) &= b \\ list\_ind(a.l; b;\ h, t, v.e) &= l[a/h,\ l/t,\ list\_ind(l; b; h, t, v.e)/v]. \end{aligned}$$

For example,

$$\begin{aligned} list\_ind(a.nil;\ 0;\ h, t, v.\ v + 1) &= list\_ind(nil;\ 0;\ h, t, v.\ v + 1) + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

One of the fundamental concepts in logic is that of an *inductively defined* type. For instance, the types of lists, terms, formulas and proofs are each defined inductively. One of the simplest inductively defined types is that of lists over a type A. Above we said that *nil* is a list; this is the *base case* of the definition. And we said that if a is in A and if l is

an A list, then $a.l$ is an A list. In general, a type is specified inductively if we show how to build an element of the type given other elements of it. Also in general there is a *base case* which does not involve using other elements of the type to build new ones; for lists it is the clause saying that *nil* is a list.

The typical informal presentation of an inductively defined set I has this pattern:

[base case:]	if b is in B then b is in I
[inductive cases:]	if $t_1, \ldots, t_n$ are in I and a is in A then $f(t_1, \ldots, t_n, a)$ is in I
	$\vdots$
	if $s_1, \ldots, s_p$ are in I and c is in C then $g(s_1, \ldots, s_p, c)$ is in I.

There is also an extremal clause saying that nothing is in I unless it is required to be there by the base and inductive clauses.

It is well-known that so called *positive inductive* definitions can be interpreted as least fixed points of monotone operators on a set (for example, see [1,24,23]). We will present only positive inductive definitions without parameters. A wider class could be defined [23], but there is no need for it here.

The syntactic style of our definitions is derived from the fixed point idea. We write the type as $rec(T.\ F(T))$ where $F(T)$ is some expression in which T occurs positively[2]. This suggests the equation $T = F(T)$. One subclass of expressions in which T occurs only positively are those built from T and other previously defined types using only the operators of cartesian product and disjoint union. For example, *A list* can be defined as $rec(T.\ \{\ nil\ \} \mid A \# T)$. The type of binary trees with leaves in a type A is $rec(T.\ A \mid T \# T)$.

One of the most important properties of inductively defined types I is that we can prove statements of the form $\forall x{:}I.\ G(x)$ by induction on I. Informally the rule for the type $I = rec(T.\ F(T))$ is this. To prove $\forall x{:}I.\ G(x)$, we assume as an induction hypothesis that we are given an arbitrary subset S of I and that $G(x)$ is true for all $x \in S$, and then show that $G(z)$ holds for all $z \in F(S)$. The arbitrary subset is presented as $\{\ v{:}I \mid P(v)\ \}$ where P is an arbitrary predicate on I.

2.2 Syntax

We now present some basic definitions which allow us to talk about terms, logical formulas, free variables, substitution of terms for variables and pattern matching of one formula against another. The basic concept is that of a term; formulas are a special case. We do not treat binding operators, such as quantifiers.

The type of terms is inductively defined. We have in mind terms such as x, $sin(x)$, $equal(x, y)$, $plus(x, y)$, $implies(x, y)$, $implies(equal(x, plus(y, 1)), equal(plus(y, 1), x))$. We say that the words *sin, equal, plus, implies* are *function symbols*. A simple definition of terms would be to say that a variable is a term, and if $t_1, \ldots, t_n$ are terms and f is a function symbol, then $f(t_1, \ldots, t_n)$ is a term. But the important point for manipulating terms is not the particular way of displaying them but rather the component parts and a means of accessing them. So for a composite term such as $f(t_1, \ldots, t_n)$, the components

[2]In the special case treated here, all occurrences are positive. But in Nuprl [8,23] a much wider class of terms is allowed.

are f and the list $(t_1, \ldots, t_n)$. The important feature of a variable is that it is the base case of the definition. So the official definition of a term is that it is a variable or it is a pair $\langle f, l \rangle$ where f is a function symbol and l is a list of terms. This is the mathematical definition, but we can agree to display a composite term as $f(l)$.

If we let *Var* be the set of variables and *Fun* the set of function symbols, then the type *Term* is defined as[3]

$$rec(\mathit{Term}.\ \mathit{Var} \mid (\mathit{Fun} \# \mathit{Term\ list})).$$

The induction principle for terms is given by the principle for this recursive type. We use it to prove the following theorem.

Theorem 1 *For all propositional functions P on terms, if $\forall x$: Var. $P(x)$, and if for all Term lists l and function symbols f, whenever $P(t)$ holds for all terms on l then $P(f(l))$, then $\forall t$: Term. $P(t)$.*

2.2.1 Term structure

We frequently carry out proofs or constructions based on the structure of a term. To facilitate them we introduce a case discriminator written as

$$case\ t:\ y \rightarrow a;\ f.l \rightarrow b.$$

This means that if the term t is a variable then the value is a, where in the expression a, y stands for the variable; if the term is composite then the value is b where f and l are names in b of the function symbol and subterm list respectively. This case analysis is used together with induction on term structure to define concepts on terms. For example, here is a definition of the notion that a variable x occurs in a term t.

Definition A variable x *occurs in* a term t, abbreviated $x \epsilon t$, if in the case that t is a variable y then $y = x$, and in the case that t is $f(l)$, then x occurs in some term u of l.

The form of inductive arguments on term structure is clarified by seeing the computational meaning of induction. The induction rule defined above is not only a way to prove properties of terms, but it is also a method of computing based on the structure of a term, just as mathematical induction is a way of computing based on the structure of natural numbers. The computational form associated with induction on terms is denoted $rec\_ind(t; h, x.\ b)$. To compute by "recursion induction" on a term t, we compute the

[3]Usually different kinds of characters are used for variables and function symbols, say $x, y, z, x_1, y_1, z_1, \ldots$ for variables and $f_1, f_2, \ldots$ for functions. We adopt the convention that any *atomic symbol* can serve as either a function or a variable. These atomic symbols come from the set *atom*. One can tell which is intended because variables occur without arguments while functions always have them. Another departure from custom is that we allow the same symbol to appear as a function or a variable and as a function taking different numbers of arguments. Thus $f(f(f), f)$ is a legitimate term, a way of writing $\langle f, (\langle f, (\langle f \rangle), f \rangle) \rangle$. This term can be interpreted in numerous ways; one possibility is that f is treated as three different objects, one a variable, one a function of one argument and one a function of two arguments. So the above term is like $f_2(f_1(f), f)$. But these are matters of meaning which we do not discuss now. We are concerned only with the *syntax* of terms.

expression b with the term t substituted for x and the entire *rec_ind* form substituted for h. Both h and x are bound variables whose scope is b. Let us consider this form for the notion of x *occurs in* t. The expression b is a case discriminator on x, precisely,

rec_ind(t; occurs, z . case z: y → x=y;
f,l → for some u on l. occurs(u))

The general rule for evaluating the recursion induction form $rec\_ind(t; h, z.\ b)$ is that it reduces to

$$b[t/z, (\lambda w.\ rec\_ind(w; h, z.\ b))/h].$$

So in the case of a term such as $f(x, v)$ the computation of the recursive definition x *occurs in f(x,v)* is the following:

rec_ind(f(x,v); occurs,z. case z: y → x=y
f,l → for some u in l. occurs(u))

reduces to (letting *occurs* denote $\lambda w.$ *rec_ind*$(w; \ldots)$)

case f(x,v): y → x=y
f,l → for some u in l. occurs(u)

which reduces to

for some u in (x,v). occurs(u).

Taking u to be x results in

case x: y → x=y
f,l → ....

In this case the term is a variable and the form reduces to

$$x = x$$

which is true. So x does occur in $f(x, v)$.

2.2.2 Substitutions

We want to study the substitution of terms for variables in terms, as in substituting 2 for x in $3 * x = z$ to obtain $3 * 2 = z$. A substitution will be defined as a list of pairs of a variable with a term, such as

$$(\langle x, 2\rangle, \langle y, z\rangle, \langle z, x + y\rangle).$$

Thus a substitution is an element of the type *Var#Term list*. Call this type *Sub*.

Given a substitution s and a variable x we write $s(x)$ to designate the term paired with x (the first if there is more than one). This notion can be defined precisely by a list induction form, namely

$$s(x) = list\_ind(s; x; h, tl, v.\ \textit{if } h.1 = x \textit{ then } h.2 \textit{ else } v).$$

(Recall that $h.1$ and $h.2$ select the first and second members of a pair respectively.)

The application of a substitution s to a term t is written $s(t)$; it is defined by induction on the structure of t. If t is a variable y, then $s(t)$ is $s(y)$. If t is f, l, then $s(t)$ is obtained by applying s to each element of the list l. If we let $map(s, l)$ denote the function that applies s to each term on the list l, then $s(t)$ is

$$rec\_ind(t;\ h, z.\ case\ z:\ y \rightarrow s(y)\ ;\ f, l \rightarrow f(map(s, l))).$$

Abbreviate $map(s, l)$ by $s(l)$.

The *domain* of a substitution is those variables that appear as the first element of a pair. We write $x \in dom(s)$ to indicate that x is in the domain of substitution s. We say that s_1 is a *sub-substitution* of s_2, written $s_1 \subset s_2$, if for every variable x such that $x \in dom(s_1)$, $x \in dom(s_2)$ and $s_1(x) = s_2(x)$. We say that a substitution is *minimal*, $min(s)$, if no variable occurs twice on the left side of a pair.

2.2.3 Matching

We are now ready to discuss the problem of (first–order) matching. This is a basic issue in automated reasoning; for instance see [6,11,2,33]. We say that term t_1 matches t_2 if there is a substitution s such that $s(t_1) = t_2$. For example, $f(x,\ g(y, z))$ matches $f(g(y, z), g(y, z))$ using the substitution $(\langle x, g(y, z)\rangle)$. But $f(x, y)$ does not match $g(x, y)$ because the outer operators are different. We want to prove that for all terms t_1 and t_2 either t_1 matches t_2, and we can find a minimal substitution s such that $s(t_1) = s(t_2)$, or no substitution will produce a match. More precisely, define $match?(t_1)$ if for every $t_2 \in Term$

$$\exists\, s{:}\, Sub.\ s(t_1) = s(t_2)\ \&\ min(s)\ \&\ \forall\, x{:}\, Var.\ x \in dom(s) \Leftrightarrow x \in t_1)$$
$$\vee\ \forall\, s{:}\, Sub.\ \neg(s(t_1) = s(t_2)).$$

The main theorem can now be stated.

Theorem 2 *(match_thm):* $\forall\, t{:}\, Term.\ match?(t)$.

The main idea of the proof is to check the outer structure of t_1 and t_2. If t_1 is a variable, then the pair $\langle t_1, t_2\rangle$ is the substitution. Otherwise t_1 is compound, say $\langle f_1, l_1\rangle$. If t_2 is a variable then there is no match, so assume it has the form $\langle f_2, l_2\rangle$. If $f_1 \neq f_2$, then there is no substitution, otherwise the result depends on the result of trying to match l_1 and l_2. Suppose l_1 is *nil*. If l_2 is also *nil* then the empty substitution matches t_1 and t_2, otherwise there is no match. Suppose l_1 is $a_1.u_1$; if l_2 is *nil* then there is no match so suppose l_2 is $a_2.u_2$. First match a_1 and a_2. If they do not match, then neither do t_1 and t_2. Suppose they do, and let s be the substitution so that $s(a_1) = a_2$. Now try recursively to match u_1 and u_2. If these do not match, then neither do t_1 and t_2. If s' matches them, then we see whether s and s' are compatible substitutions, *i.e.* whether they agree on common variables. If they do, then the final substitution is a union of s and s'. If they are incompatible, then we must argue that no match exists.

2.3 Semantics

One account of the meaning of sentences in formal logic is that they denote a *truth value*. The meaning of operations that combine sentences, such as *and, or, implies* and *not* can in this account be explained as operations on truth values. For example, P and Q is a true sentence if P is true and Q is true; whereas it is false when one of P or Q is false. We develop next a brief account of the semantics of the propositional connectives $\&$, $\vee$, $\Rightarrow$ and $\neg$. Syntactic matters are covered completely by treating these operators as function symbols. The terms built from these operators are called *propositional formulas*. For example, the *contrapositive law* of logic usually written

$$(p \Rightarrow q) \Rightarrow (\neg q \Rightarrow \neg p)$$

is written as the term

$$imp(imp(p, q),\ imp(not(q), not(p))).$$

To assert this as a law is to say that its truth value is *true* regardless of the values of the variables p, q which stand for arbitrary propositions. We will express this precisely by defining a function that takes as arguments a propositional formula and an assignment of truth values to the variables of the formula, and returns the truth value of the formula given the assignment. An assignment is simply a substitution in which each term is a constant, either *true* or *false*. That is,

$$Assignment \ = \ \{\, s{:}\,Sub \mid for\ all\ p\ in\ s.\ p.2 = true \vee p.2 = false \,\}.$$

The value of a formula f under a truth assignment $a \in Assignment$, written $value(a, f)$, is

$$\begin{aligned} case\ f\ ;\ x \rightarrow a(x)\ ;\ op, l \rightarrow case\ op :\ & and \rightarrow value(a, l.1)\ \&\ value(a, l.2) \\ & or \rightarrow value(a, l.1) \vee value(a, l.2) \\ & imp \rightarrow \neg value(a, l.1) \vee value(a, l.2) \\ & not \rightarrow \neg value(a, l.1), \end{aligned}$$

where $l.1$ and $l.2$ here are the first and second elements of the list l, and the operators $\&$, $\vee$, $\Rightarrow$ and $\neg$ are functions computing the appropriate truth values. A formula f is *valid* or a *tautology* if for every assignment a, $value(a, f) = true$. An assignment a *falsifies* f if $value(a, f) = false$.

2.3.1 Decision procedure for validity

We want to show that we can decide for any propositional formula whether or not it is valid. Moreover, we want the computational content of the argument to be a special algorithm called a *tableau decision* procedure [31,16]. One elegant way to express the procedure is in terms of a pair of formula lists called a *sequent*.

$$Sequent \ = \ (Term\ list)\#(Term\ list)$$

If H and G are lists of terms, then we sometimes write the sequent containing them as $H \gg G$, calling H the *hypothesis list* and G the *goal list*. We extend the notions of truth

and validity to sequents by taking $H \gg G$ to be true if the conjunction of the elements of H implies the disjunction of those of G. This can be made formal with the following definitions.

$$\begin{aligned} \mathit{conj}(H) &= \mathit{list\_ind}(H;\ \mathit{true};\ h, t, v.\ \mathit{and}(h, v)) \\ \mathit{disj}(H) &= \mathit{list\_ind}(H;\ \mathit{false};\ h, t, v.\mathit{or}(h, v)) \\ \mathit{svalue}(a, \langle H, G\rangle) &= \mathit{value}(a,\ \mathit{imp}(\mathit{conj}(H), \mathit{disj}(G))) \\ \langle H, G\rangle\ \mathit{is\ valid} &\Leftrightarrow \forall\, a{:}\,\mathit{Assignment}.\ \mathit{svalue}(a, \langle H, G\rangle) = \mathit{true} \end{aligned}$$

With these definitions, the exact result we want is:

$$\forall\, s{:}\,\mathit{Sequent}.\ \mathit{valid}(s) \vee \exists\, a{:}\,\mathit{Assignment}.\ (\mathit{svalue}(a, s) = \mathit{false})$$

The method of proof will be to decompose s into smaller sequents which have the property. We show that if the smaller sequents are valid, so is the original, and if the smaller sequents are falsifiable, then so is the original. Here follows a typical step of the argument.

For example, given a sequent $H \gg (A \Rightarrow B)$ we form the smaller sequent $A, H \gg B$ If this is valid, then so is the original. If assignment a falsifies this smaller sequent, then it gives value *true* to A and all elements of H and *false* to B. But such an assignment will falsify $H \gg (A \Rightarrow B)$ because it assigns *true* to H and *false* to $(A \Rightarrow B)$.

One of the key features of the argument is the definition of a *smaller* sequent. Notice that in going from $H \gg (A \Rightarrow B)$ to $A, H \gg B$ one occurrence of an operator has been eliminated. Thus the "over-all number of operators" is decreasing. We write $s' < s$ if s' has fewer operators than s.

Theorem 3 *(Validity is Decidable):*

$$\forall\, s{:}\,\mathit{Sequent}.\ \mathit{valid}(s) \vee \exists\, a{:}\,\mathit{Assignment}.\ (\mathit{svalue}(a, s) = \mathit{false})$$

Proof. The proof is by induction on the number of operators in s. Let s be an arbitrary sequent, say $s = \langle H, G\rangle$. The induction hypothesis is:

$$\forall\, s'{:}\,\mathit{Sequent}.\ s' < s \Rightarrow P(s').$$

Suppose that all formulas of s are atomic (so there are no s' where $s' < s$) Either H and G are disjoint or there is some formula f in both H and G. In the case that H and G are disjoint, there is an assignment which assigns *false* to all variables of G and *true* to all variable of H. Under this assignment $\mathit{svalue}(a, s) = \mathit{false}$ by definition of *svalue*. If f occurs in both H and G, then $\langle H, G\rangle$ is valid because each assigument that assigns value *true* to f in H will make $\mathit{disj}(G)$ true as well.

Now suppose that there is at least one formula f of H or G which is compound. The argument proceeds by cases on the *location* and *structure* of f. Let us assume first that the goal G has a compound formula, and that f is the first compound formula (from the left)

G1. If f is $f_1 \vee f_2$ then form G' from G by replacing f by f_1, f_2. Let $s' = \langle H, G'\rangle$ and notice that $s' < s$, so that by the induction hypothesis the theorem holds for s'. If s' is valid, then clearly s is as well, and if a falsifies s' then it falsifies s as well.

G2. If f is $f_1 \& f_2$ then form G_1 by replacing f by f_1 and form G_2 by replacing f by f_2. Let $s_1 = \langle H, G_1 \rangle$ and $s_2 = \langle H, G_2 \rangle$. Notice that $s_1 < s$ and $s_2 < s$, so the theorem is true s_1 and s_2.

If both s_1 and s_2 are valid, then clearly s is as well. If one of them, without loss of generality say s_1, is falsifiable using a, then a falsifies s as well.

G3. If f is $(f_1 \Rightarrow f_2)$ then replace f in G by f_2 to obtain G' and add f_1 to H to form H'. Let $s' = \langle H', G' \rangle$. Suppose s' is valid, and suppose that the assignment a makes H true. If $a(f_1) = \mathit{false}$ then G is true, so suppose $a(f_1) = \mathit{true}$. H' must then be true. Since s' is valid G' must be true under a hence so must G.

If s' is falsifiable by a, then the same a falsifies s because f_1 is true and f_2 is false, so $(f_1 \Rightarrow f_2)$ is false along with all the other formulas of G.

G4. If f is $\neg f_1$ then replace f in G by *false* to form G' and add f_1 to H to form H'. Let $s' = \langle H', G' \rangle$. We can now argue as in the previous case.

Now suppose there is *no* compound formula in G, but there is one in H, say f. Consider the possible forms of f.

H1. If f is $f_1 \vee f_2$ then replace f by f_1 in H to form H_1 and by f_2 to form H_2. Let $s_1 = \langle H_1, G \rangle$ and $s_2 = \langle H_2, G \rangle$. Notice that $s_1 < s$ and $s_2 < s$, so the theorem is true for s_1 and s_2. If both s_1 and s_2 are valid, then so is s. If either s_1 or s_2 is falsified by a, then the same a falsifies s.

H2. If f is $f_1 \& f_2$ then replace it in H by f_1, f_2 to form H' and let $s' = \langle H', G \rangle$. If s' is valid, then clearly s is as well. If s' is falsifiable by assignment a, then a falsifies s since the truth values of H and H' are the same.

H3. If f is $(f_1 \Rightarrow f_2)$ then replace f by f_2 to create H' and add f_1 to G to create G'. Let $s_1 = \langle H', G \rangle$ and $s_2 =< H'', G' >$ where H'' is H without f. If both s_1 and s_2 are valid, then we argue that s is valid. Consider any assignment which makes H true. Then f is made true which means either f_2 is true or both f_1 and f_2 are false. If f_2 is true then so is H' so from the validity of s_1 we know that G is true. If both f_1 and f_2 are false, then since s_2 is valid and H'' is true, some element of G must be true.

If one of s_1 or s_2 is falsifiable by a, then s will be. Suppose s_1 is falsifiable. Then f_2 and hence f will have value *true* under a, but G will be false, so s is false. On the other hand, if s_2 is falsifiable by a, then f_1 has the value *false*. But then in s, f will get value true and all formulas of G will remain false.

H4. If f is $\neg f_1$ then remove f from H to obtain H' and add f_1 to G to obtain G'. Let $s' = \langle H', G' \rangle$. If s' is valid, then s is as well, that is, when $\neg f_1$ gets the value *true* in H, then f_1 gets the value *false*, so in G' some other formula must get the value *true*, and this is the formula true in G. If s' is falsifiable by a, then H' is true but G' is not. Thus f_1 gets the value *false*, so H is true and all formulas of G remain false.

Qed.

Notice that the proof of this theorem is constructive. It implicitly gives an algorithm to decide validity, and the algorithm is the tableau procedure. In the next section we show how to formalize the theorem on matching in the constructive type theory of Nuprl. The formal library follows very closely the informal development.

3 Formal Metamathematics

A system that supports our approach to automatic theorem proving should have several properties. First, it should be based on a rich formal logic that allows direct expression of metamathematical ideas. Since we are interested in the computational content of metamathematics, the logic should be constructive, so that, for example, in asserting that a theory is decidable we are also implicitly asserting that there is a procedure for deciding when a given sentence is true. We want to apply the computational content of our metamathematics in the construction of proofs, so the system should be able to extract the algorithms that are implicit in our theorems.

Since programs are to be extracted from metatheorems, and since we must be certain that these programs are correct so that logical soundness is preserved, the metatheorems must be formally proven. Since programs are determined by proofs, we will often want to construct formal proofs that follow a specific outline. For example, in proving that any propositional formula is either valid or not, we gave an argument that implicitly constructs a tableau decision procedure. Ideally we would like to be able to construct the formal proof by supplying the proof we want at as high a level as possible, with the system taking care of the detailed reasoning necessary to complete the proof.

The Nuprl proof development system[8] goes a good part of the way toward satisfying the above properties. Nuprl's logic was designed as a foundation for constructive mathematics, and there is little doubt that it is sufficiently expressive for the applications we have in mind. By design it has an extraction property; complete proofs yield programs in a direct and natural manner. The extent to which Nuprl allows direct expression of and high-level natural reasoning about metamathematics is difficult to quantify. We will not do so, but will instead give an extended example of formal metamathematics in Nuprl. In particular, most of the rest of this section is devoted to describing a formal proof we have carried out in Nuprl of the theorem about matching that was discussed earlier.

The formal proof, which was developed specifically for this paper, took about a day to construct using Nuprl. A complete listing of the proof can be obtained from the authors. The actual Nuprl-readable version is available with the distribution of the Nuprl system.

Before we can proceed with the example, we need to give a brief description of Nuprl. After the example, we discuss how programs such as matching can be incorporated in Nuprl's own inference mechanisms.

3.1 Nuprl

Nuprl [8] is a system that has been developed at Cornell by a team of researchers, and is intended to provide an environment for the solution of formal problems, especially those

where computational aspects are important. One of the main problem-solving paradigms that the system supports is that of *proofs-as-programs*, where a program specification takes the form of a mathematical proposition implicitly asserting the existence of the desired programs. Such a proposition can be formally proved, with computer assistance, in a manner resembling conventional mathematical arguments. The system can extract from the proof the implicit computational content, which is a program that is guaranteed to meet its specification.

The logical basis of Nuprl is a constructive type theory that is a descendent of a type theory of Martin-Löf [22]. We will not give details of the type theory except as necessary in the discussion of the match example, and in particular we will not discuss how notions of logic are expressed in the type theory (but we used informally the basic concepts of this type theory in section 2.2). Higher-order logic is defined directly in terms of types and can be used abstractly. When we refer to "formulas" or "propositions" below we mean these defined notions, and often the statements we make about formulas will be true of types in general. The following account is somewhat loose; a complete account of the type theory and system is contained in the Nuprl book [8].

The inference rules of Nuprl deal with *sequents*, which are objects of the form

$$H_1,\ H_2,\ \ldots,\ H_n >>\ P$$

where P is a formula and where each H_i is either a formula or a variable declaration of the form x: T for x a variable and T a type. The H_i are referred to as *hypotheses*, and P is called the *conclusion* of the sequent. Sequents, in the context of a proof, are also called *goals*. A sequent is true if the conclusion is true whenever the hypotheses are. We take truth here to be constructive, so that a true sequent comes with a procedure giving its computational content. An important point about the rules of Nuprl is that they preserve constructive truth, so that from a complete proof of a sequent $>>P$ the computational content of P can be computed.

A proof in Nuprl is a tree-structured object where each node has associated with it a sequent and a rule. We call rules in Nuprl *refinement* rules because we think of them as being applied backwards: given a goal, we *refine* it by using an inference rule whose conclusion matches the goal, obtaining *subgoals* which are the premises of the rule. The children of a node in a proof tree are the subgoals which result from the application of the refinement rule of the node. A key feature of Nuprl's proof structure is that a refinement rule need not be just a primitive inference rule, but can also be a *tactic* written in the programming language ML [14]. As with a primitive refinement rules, the application of a tactic to a goal produces subgoals. Because of ML's strong typing property, it is guaranteed that the subgoals imply the goal. When a tactic is used to refine a goal, the rule at the new node of the proof tree will show the text of the tactic. This gives a means for constructing higher level proofs that can serve as explanations of formal arguments.

Suppose we have proven a theorem

```
∀i,j,m,n: Int. i≤j & m≤n => m+i≤n+j
```

and named it mono. An example of a refinement step using a tactic which applies this

theorem is the following.[4]

```
x:Int, 0≤x  >>  x+2 ≤ 2*x+2   BY (Lemma 'mono' ...)
   x:Int, 0≤x  >>  x ≤ 2*x
```

The tactic **Lemma** computes the necessary terms and applies the rules necessary to instantiate the theorem. The three dots after the tactic indicate that a general purpose tactic called the *autotactic* was applied after **Lemma**. In this example, the autotactic proved the most trivial subgoals produced by **Lemma** (that **x**, **2*x** and **2** are integers and that $2 \leq 2$). The result of the refinement is the single subgoal shown.

Interactions with Nuprl are centred on the *library*. A library contains an ordered collection of definitions, theorems, and other objects. New objects are constructed using special-purpose window-oriented editors. The text editor, together with the definition facility, permit very readable notations for objects of Nuprl's type theory. The proof editor is used to construct proofs in a top-down manner, and to view previously constructed proofs. Proofs are retained by the system, and can be later referred to either by the user, or by tactics.

3.2 A Formal Account of Matching

In Section 2.2 we gave an informal type-theoretic account of first-order matching. We have implemented a formalization of this account in Nuprl. The complete Nuprl library containing this formalization can be divided into two parts. The first part, consisting of about 150 objects, is of a general nature. It contains definitions and simple theorems pertaining to the representation of logic within Nuprl. It also contains a minimal development of the theory of lists. The second part is particular to the formalization of matching; it contains about 40 definitions and theorems, most of which concern substitution and the syntax of terms.

The description that follows consists of three parts. First is a brief discussion of the tactics that were used in building the proofs in this library. Next is a presentation of the definitions and the simple theorems leading up to the main results of the library. Last is a detailed description of the proofs of the three main theorems of the library. In particular, we will examine one of the proofs in its entirety in order to give an idea of the character of reasoning about formal metatheory in Nuprl. These three parts constitute a complete description of the library, in that all definitions and theorems not in the general portion of the library will be at least stated.

The definitions, theorems and proof steps that we show below appear much as they would on the screen of a Nuprl session, although a few liberties have been for the sake of readability. In particular, a few variables were renamed, and a quirk of the system whereby display forms associated with definitions were occasionally lost was manually corrected for. Also, the syntax of Nuprl definitions was slightly altered for compactness. The notations for defined terms, including the special symbols for quantification, *etc.*, are exactly as they appear in the system.

[4] We will use typewriter typeface when presenting objects constructed (or hypothetically constructed) with the Nuprl system.

3.2.1 Preliminaries

A tactic in Nuprl is an ML program that, when applied to a sequent, either *fails* or returns a list of subgoal sequents. Over the last several years a rather large collection of tactics has been built up at Cornell. Some of these are described in [17]. Only a few of these need to be described here. Little detail will be given, but it should be enough so that their use in the proof steps we give later will be understandable.

The bulk of the work in proving the theorems in our library, in terms of number inference steps taken, is done by two general tactics. The first of these, the *autotactic*, was mentioned above. The autotactic is usually able to completely prove subgoals involving typechecking (showing that a term is in a certain type or that a formula is well-formed—these properties are recursively undecidable in Nuprl because of the generality of the type theory) and simple kinds of simple integer arithmetic and propositional reasoning. Associated with definitions in the library are theorems which give a type for the defined term; these types are used by the autotactic in proving the numerous typechecking subgoals that arise. The autotactic is usually effective in hiding the details of the type theory from the user; for example, there are no typechecking subgoals in the proofs of the main theorems described below. The autotactic is usually invoked via a Nuprl definition: when (T ...) appears in a proof, it means that the autotactic was applied after the tactic T.

The other general tactic is called `Simp` and is used to simplify terms. This tactic can be updated from the library (through a special kind of library object that can contain an ML form) with new simplification clauses in two basic ways. First, one can specify that a theorem be used for simplification. The theorem should be a universally quantified formula of the form

$$A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_n \Rightarrow R(a, b)$$

where R is an equivalence relation (such as if-and-only-if or Nuprl's built-in equality). The simplifier will then always try to rewrite terms matching a to an instance of b. Second, one can directly specify a simplification that is computationally justified (where one term can be obtained from another by a sequence of forward or backward computation steps). For example, one can specify that the length `|h.1|` of a list with head `h` always be simplified to `|1|+1`. The "general" portion of the match library contains numerous updates to the simplifier, mostly for propositions and some for list theory. Like the autotactic, the simplifier is invoked by a definition: the notation (T ...`+s`) indicates that the simplifier and autotactic were both applied after T. One of the variants of this has `sc` in place of `+s`; this means the simplifier was applied to the just the conclusions of the subgoals produced by T.

Several other tactics use information from the library. `Induction` is used to perform induction on a variable appearing in a goal. If there is an induction principle in the library for the type of the variable, that is used, otherwise the type should be a primitive one (not a definition instance) for which there is an induction rule. `Unroll` is similar to induction except that no induction hypothesis is generated. Finally, `Decide` takes a formula P and performs a case analysis on whether or not P is true. Decidability ($P \vee \neg P$) is non-trivial in Nuprl, and `Decide` attempts to use theorems of the appropriate form to justify the case analysis.

There are quite a few tactics for purely logical reasoning; most of these are simple.

For example, `ILeft` reduces proving a disjunction to proving the left disjunct, and `ITerm` reduces proving an existentially quantified term $\exists x\colon T.\ P(x)$) to proving $P(t)$ for some supplied term t. ("I" is for introduction). For analysing hypotheses there are various "elimination" tactics: `E` applies to most kinds of formula and performs one step of analysis; `SomeE` applies to "some" (or "exists") formulas; and `EOn` (which requires a term argument) applies to universally quantified formulas. `HypCases` uses a universally quantified disjunction in the hypothesis list to perform a case analysis; it takes a list of terms as arguments to instantiate the quantified variables with. The most powerful tactic for logical reasoning is `Backchain` which uses (universally quantified) implications in the hypothesis list in a search for a complete proof. For example, if the conclusion of a sequent is Q and $P \Rightarrow Q$ is a hypothesis, then it reduces proving Q to proving P and then attempts to recursively prove Q; if this fails it tries another hypothesis of a similar form.

There are several simple tactics for lemma application. One of these is `Lemma`, an example application of which was given above. `FLemma` does a forward-reasoning analogue of what `Lemma` does. For example, if the theorem referred to is $P \Rightarrow Q$ and P is a (specified) hypothesis, then Q is added as a new hypothesis. Finally, `CaseLemma` uses a named lemma to determine a case analysis.

Finally, there are a few miscellaneous tactics that should be mentioned. First, the tactic `Thin` and its variants are used to remove unwanted hypotheses from a goal. `AndThin` takes a tactic that applies to hypotheses, applies it to a hypothesis and then discards the hypothesis. The so-called *tactical* `THEN` is used to combine tactics; if T_1 and T_2 are tactics then the tactic T_1 `THEN` T_2 first applies T_1 then applies T_2 to the resulting subgoals. The last tactic we discuss is `Expand`. This tactic takes a list of names of definitions and expands all instances of those definitions in the goal.

3.2.2 Definitions and Simple Theorems

Since the definitions given in Section 2.2 are given in type-theoretic terms, their formalizations are almost identical. In particular, the explanations given there still apply, so we will do little more here than to simply present the formal versions. In Nuprl a definition generally consists of two objects. One object is used to establish the defined term (and usually to give it a type), and the other is used to attach a notation or display form to the definition. For brevity we will present definitions by collapsing these two objects into one, just giving the notation and the term it denotes.

The simple theorems given below were easy to prove, and we will only present a proof of one of them. These theorems required on average about five steps each to prove. What a "step" is is not well-defined since we can always collapse an entire proof into a single step by composing all the tactics used in the proof. We will rely on our description of some representative proofs to convey what a typical step is in this setting.

The definitions for syntax are straightforward. We first define the types of representatives of functions and variables in terms of Nuprl's type of atoms (character strings).

```
Var == Atom
Fun == Atom
```

The type of term representatives is a recursive type of syntax trees:

```
Term == rec(T. Var | Fun # T list).
```

The equality relations for terms and lists of terms are frequently used, so definitions are made for them.

```
t1=t2 == t1=t2 in Term
l1=l2 == l1=l2 in Term list
```

Note that ambiguities in notations are acceptable in Nuprl (since definition instances do not have to be parsed). The "injections" or "term constructors" are defined by

```
x == inl(x)
f(l) == inr(<f,l>)
```

We prove an induction principle and a useful special case of it:

```
>> ∀P:Term->U1. (∀x:Var. P(x))
             => (∀l:Term list. ∀f:Fun. (∀t:l. P(t)) => P(f(l)))
             => ∀t:Term. P(t)
>> ∀P:Term->U1. ∀x:Var. P(x)
             => ∀l:Term list. ∀f:Fun. P(f(l))
             => ∀t:Term. P(t)
```

Here the proposition `∀t:l. P(t)` asserts that `P(t)` is true for every member `t` of the list `l`. The first of these theorems can be read as "The property P of terms is true for all terms if (1) it is true for all variables and (2) if f is a member of Fun and l is a list of terms satisfying P then P is true of $f(l)$". For our purposes, the type `U1` can be thought of as the type of all propositions.

The form for case analysis of terms is the following.

```
case t: x→a; f,l→b == decide(t; x.a; ap. let f,l=ap in b)
```

This uses the type theory's operators for analyzing a member of a disjoint union and for decomposing a pair. As in Section 2.2, we can now define when a variable *occurs* in a term.

```
x∈t == rec_ind(t; P,z. case z: y → y=x; f,l→ ∃u:l. P(u))
```

We also define a version of this predicate for lists

```
x∈l == ∃t:l. x∈t
```

so that a variable occurs in a list if it occurs in some member of the list. For this definition as with other defined recursive functions and predicates, we add appropriate clauses to the simplifier. For example, we want any term of the form `x∈f(l)` to simplify to `x∈l`.

The type of substitutions and the application of a substitution to a variable and a term are defined as before. We also define application to a list of terms.

```
Sub == Var#Term list
s(x) == list_ind(s; x; h,ll,v. if h.1=x then h.2 else v)
s(t) == rec_ind(t; h,z. case z: x→s(x); f,l→f(map(h,l)))
s(l) == map(λt.s(t), l)
```

As in Section 2.2, the notations .1 and .2 denote projections from a pair, and map(h,l) applies the function h to every element of the list l.

We will need several predicates on substitutions. In particular, we define when a variable is in the domain of a substitution, when one substitution is contained in another, when a substitution is minimal (that is, no two pairs in it have the same variable component) and when two substitutions are inconsistent (that is, disagree on some variable in both their domains).

```
x∈dom(s) == ∃p:s. p.1=x
s1⊂s2 == ∀x:Var. x∈dom(s1) => x∈dom(s2) & s1(x)=s2(x)
min(s) == list_ind(s; True; h,l,v. ¬(h.1∈dom(l)) & P)
ncst(s1,s2) == ∃x:Var. x∈dom(s1) & x∈dom(s2) & ¬(s1(x)=s2(x))
```

Finally, it will be handy to have the following definition for the main theorem.

```
match?(t1) == ∀t2:Term. ∃s:Sub. s(t1)=t2 & min(s)
                                & ∀x:Var. x∈dom(s) <=> x∈t1
                          ∨ ∀s:Sub. ¬(s(t1)=t2)
```

Thus match?(t1) asserts that for every t2 either there is a minimal matching substitution or there is no matching substitution. The minimality and the condition that a variable be in the domain of the substitution exactly if it occurs in t are required for our inductive proof to go through. We also have a version of the above definition for lists.

```
match?(l1) ==  ∀l2:Term list. ∃s:Sub. s(l1)=l2 & min(s)
                                    & ∀x:Var. x∈dom(s) <=> x∈l1
                              ∨ ∀s:Sub. ¬(s(l1)=l2)
```

This list version is defined simply for convenience, as will be made clear when we present the proof of the main theorem.

The following seven simple lemmas are incorporated into the simplifier and never have to be explicitly referred to. The fifth of these may look trivial; this is because we have used the same display form for term equality as for equality in Var.

```
>> ∀x,y:Var. ∀s:Sub. ∀t:Term. x=y => (<x,t>.s)(y) = t
>> ∀x,y:Var. ∀s:Sub. ∀t:Term. ¬(x=y) => (<x,t>.s)(y) = s(y)
>> ∀x,y:Var. ∀s:Sub. ∀t:Term. x=y => x∈dom((<y,t>).s) <=> True
>> ∀x,y:Var. ∀s:Sub. ∀t:Term. ¬(x=y) => x∈dom((<y,t>).s) <=> x∈dom(s)
>> ∀x,y:Var. x=y <=> x=y
>> ∀f1,f2:Fun. ∀l1,l2:Term list. f1(l1)=f2(l2) <=> f1=f2 & l1=l2
>> ∀x:Var. ∀f:Fun. ∀l:Term list. x=f(l) <=> False
```

The following is the first theorem we prove that has an interesting computational interpretation. Constructively it means that we can decide whether or not two terms are equal. From the proof of this theorem Nuprl can extract a decision procedure.

```
>> ∀t1,t2:Term. t1=t2 ∨ ¬(t1=t2)
```

```
* top
>> ∀x:Var. ∀s:Sub. ¬(x∈dom(s)) ∨ ∃t:Term. x∈dom(s) & s(x)=t

BY (On 's' Induction ...+s)

1* 1. x: Var
   2. s: Sub
   3. p: Var#Term
   4. ¬(x∈dom(s)) ∨ ∃t:Term. x∈dom(s) & s(x)=t
   >> ¬(p.1=x ∨ x∈dom(s))
      ∨ ∃t:Term. (p.1=x ∨ x∈dom(s)) & (p.s)(x)=t
```

Figure 1: Proof by induction on s.

The next two theorems establish that the value of a substitution on a term (list) is characterized by its values on the variables occurring in the term (list).

```
>> ∀s1,s2:Sub. ∀t:Term.
    s1(t)=s2(t) <=> (∀x:Var. x∈t => s1(x)=s2(x))
>> ∀l:Term list. ∀s1,s2:Sub.
      s1(l)=s2(l) <=> (∀x:Var. x∈l => s1(x)=s2(x))
```

The computational content of the next simple theorem is a procedure which "looks up" a binding for a variable x in a substitution s, producing an indication of whether or not the variable is in the domain of s and in the former case giving the term bound to x.

```
>> ∀x:Var. ∀s:Sub. ¬(x∈dom(s)) ∨ ∃t:Term. x∈dom(s) & s(x)=t
```

For future reference, the name of this theorem is sub_lookup. The proof is short and is given in its entirety as Figures 1 to 4. The figures show the proof steps as they would appear to a Nuprl user except that to save space we have removed from some steps some of the hypotheses that appear in earlier steps.

The first step in the proof is in Figure 1 and is by induction on the list s. This figure shows the goal sequent, then the rule applied, in this case a tactic, and then the subgoal sequent with its hypothesis list numbered and displayed vertically. The asterisk at the top left is Nuprl's indication that the proof is complete, and top is the tree address of the proof node within the proof tree for the theorem. Note that in this step the simplifier and autotactic together automatically proved the base case and left the remaining subgoal in a simplified form. The step applied to this subgoal is shown in Figure 2. Here we decompose ("eliminate") the pair p (and thin, or discard, its hypothesis) and then perform some reduction steps to simplify the subgoal. The next step is shown in Figure 3. In this goal we know that the property we are proving is inductively true of s, and we have to prove it for <x1,t1>.s. We want to do a case analysis on whether x1=x, so we use the tactic Decide. The negative case was proved automatically, and we are left with a simple subgoal in the other case. The last step (Figure 4) is trivial.

```
* top 1
3. p: Var#Term
4. ¬(x∉dom(s)) ∨ ∃t:Term. x∉dom(s) & s(x)=t
>> ¬(p.1=x ∨ x∉dom(s))
   ∨ ∃t:Term. p.1=x ∨ x∉dom(s) & p.s(x)=t

BY (AndThin E 3 THEN Reduce ...)

1* 4. x1: Var
   5. t1: Term
   >> ¬(x1=x ∨ x∉dom(s))
      ∨ ∃t:Term.  (x1=x ∨ x∉dom(s)) & (<x1,t1>.s)(x)=t
```

Figure 2: Let x1 and t1 be the components of p.

```
* top 1 1
3. ¬(x∉dom(s)) ∨ ∃t:Term. x∉dom(s) & s(x)=t
4. x1: Var
5. t1: Term
>> ¬(x1=x ∨ x∉dom(s))
   ∨ ∃t:Term.  (x1=x ∨ x∉dom(s)) & (<x1,t1>.s)(x)=t

BY (Decide 'x1=x' ...+s)

1* 5. x1=x
   6. ¬(x∉dom(s)) ∨ ∃t:Term. x∉dom(s) & s(x)=t
   >> ∃t:Term. t1=t
```

Figure 3: Case analysis on whether or not x1=x.

```
* top 1 1 1
5. x1=x
6. ¬(x∉dom(s)) ∨ ∃t:Term. x∉dom(s) & s(x)=t
>> ∃t:Term. t1=t

BY (ITerm 't1' ...)
```

Figure 4: Take t to be t1.

The structure of the program extracted from this proof is determined by the steps we have shown. The first step (using induction) introduces list recursion. In the next step, the induction hypothesis corresponds to a recursive call of the function we are constructing. To lookup the value of x in p.s we first decompose the pair p into its components x1 and t1. The use of Decide in the next step corresponds to the introduction of an "if-then-else" expression whose condition is x1=x. In the case where x1 is not equal to x, a recursive call is made (that is, the induction hypothesis is used, automatically). In the other case we return the value t1.

The last theorems in our library before the three main theorems (which are discussed below) are added to the simplifier and forgotten.

```
>> ∀s1,s2:Sub. ∀x:Var. ∀t:Term.
     s1⊂s2 => (<x,t>.s1 ⊂ <x,t>.s2 <=> True)
>> ∀s1,s2:Sub. ∀x:Var. ∀t:Term.
     s1⊂s2 => ¬(x∉dom(s1)) => (s1 ⊂ <x,t>.s2 <=> True)
>> ∀s1,s2:Sub. ∀x:Var. ∀t:Term.
    s1⊂s2 => x∉dom(s2) => s2(x)=t => (<x,t>.s1 ⊂ s2 <=> True)
```

3.2.3 Matching

The match procedure is mostly the product of the last three theorems in the library.

```
>> ∀s1,s2:Sub. min(s1) & min(s2)
     => ncst(s1,s2)
        ∨ ∃s:Sub. min(s) & s1⊂s & s2⊂s
                  & ∀x:Var. x∉dom(s) => x∉dom(s1) ∨ x∉dom(s2)
>> ∀l:Term list. (∀t:l. match?(t)) => match?(l)
>> ∀t:Term. match?(t)
```

The names of these theorems in the library are sub_union, lmatch_thm and match_thm respectively. The proof of the first theorem has 23 steps, the proof of the second has 25, and the last has 13. Space limitations prevent us from presenting all of these proofs, so we will instead give the complete proof of the last theorem and just give some of the highlights of the other two. The level of inference in the three proofs is roughly the same.

The proof of match_thm is given in Figures 5 to 17. This proof is rather self explanatory, so not much additional explanation will be given. The first step (Figure 5) is by induction on t. The proof of the base case (where t is a variable) is easy and is contained in Figures 6 and 7. The first step in the proof of the induction step (Figure 8) is to apply the second of the three "main" theorems listed above, and then (Figure 9) we expand the definition of match?. In the next step (Figure 10) we "unroll" t2, doing a case analysis on whether t2 is a variable or an application. The variable case is easy (Figure 11). In the other case (Figure 12) we do a case analysis on whether the function parts of the two terms are the same.

The case where they are not the same is proved in one step (Figure 13). When they are the same we need to know if one argument list matches the other (Figure 14). When

```
* top
>> ∀t:Term. match?(t)

BY (On 't' Induction ...)

1* 1. x: Var
   >> match?(x)

2* 1. l: Term list
   2. f: Fun
   3. ∀t:l. match?(t)
   >> match?(f(l))
```

Figure 5: Proof by induction on t.

```
* top 1
1. x: Var
>> match?(x)

BY (Expand ''matchp'' ...+s)

1* 1. x: Var
   2. t2: Term
   >> ∃s:Sub. s(x)=t2 & min(s) & ∀x1:Var. x1∈dom(s) <=> x=x1
      ∨ ∀s:Sub. ¬(s(x)=t2)
```

Figure 6: Use the definition of match?.

it does, we get the required matching substitution (Figure 15). Otherwise, we obtain a contradiction (Figures 16 and 17).

We finish our presentation of the formalization of the matching algorithm by briefly discussing the other two major theorems of the library. For one we will just describe the computational content of its proof, and for the other we will show the steps of the proof that are computationally the most important.

The theorem sub_union says that any two minimal substitutions s1 and s2 are either *inconsistent* or can be combined into a minimal subsitution (the "union"). We limit our discussion of the proof of this theorem to describing the algorithm embodied in it. The union of the empty list and s2 is s2. The union of <x,t>.s1 and s2 is computed as follows. By sub_lookup (whose proof was given above), x is either in the domain of s2, in which case s2(x)=t2 for some t2, or it is not. In the first case, if t is not equal to t2 then the substitutions are inconsistent, otherwise the result is just the recursively computed union of s1 and s2. In the second case, the result is <x,t>.s where s is the union of s1 and s2.

```
* top 1 1
1. x: Var
2. t2: Term
>> ∃s:Sub. s(x)=t2 & min(s) & ∀x1:Var. x1∈dom(s) <=> x=x1
   ∨ ∀s:Sub. ¬(s(x)=t2)

BY (ILeft THENM ITerm '[<x,t2>]' ...+s)
```

Figure 7: The left disjunct is true: take s to be [<x,t2>].

```
* top 2
1. l: Term list
2. f: Fun
3. ∀t:l. match?(t)
>> match?(f(l))

BY (FLemma 'lmatch_thm' [3] ...) THEN Thin 3

1* 3. match?(l)
   >> match?(f(l))
```

Figure 8: By lmatch_thm and 3 we have match?(l).

```
* top 2 1
1. l: Term list
2. f: Fun
3. match?(l)
>> match?(f(l))

BY (Expand ''matchp'' ...+s)

1* 3. t2: Term
   4. match?(l)
   >> ∃s:Sub. f(s(l))=t2 & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(l))=t2)
```

Figure 9: Use the definition of matchp.

```
* top 2 1 1
3. t2: Term
4. match?(l)
>> ∃s:Sub. f(s(l))=t2 & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(l))=t2)

BY (On 't2' Unroll ...)

1* 4. x: Var
   >> ∃s:Sub. f(s(l))=x & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(l))=x)

2* 4. l2: Term list
   5. f2: Fun
   >> ∃s:Sub. f(s(l))=f2(l2) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(l))=f2(l2))
```

Figure 10: Case analysis on the term kind of t2.

```
* top 2 1 1 1
4. x: Var
>> ∃s:Sub. f(s(l))=x & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(l))=x)

BY (IRight ...+s)
```

Figure 11: In this case there are no matching substitutions.

```
* top 2 1 1 2
4. l2: Term list
5. f2: Fun
>> ∃s:Sub. f(s(l))=f2(l2) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(l))=f2(l2))

BY (Decide 'f=f2' ...)

1* 6. f=f2
   >> ∃s:Sub. f(s(l))=f2(l2) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(l))=f2(l2))

2* 6. ¬(f=f2)
   >> ∃s:Sub. f(s(l))=f2(l2) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(l))=f2(l2))
```

Figure 12: Either f=f2 or not.

```
* top 2 1 1 2 2
3. match?(l)
4. l2: Term list
5. f2: Fun
6. ¬(f=f2)
>> ∃s:Sub. f(s(l))=f2(l2) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(l))=f2(l2))

BY (IRight ...+s)
```

Figure 13: Clearly no match in this case.

```
* top 2 1 1 2 1
3. match?(1)
6. f=f2
>> ∃s:Sub. f(s(1))=f2(12) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(1))=f2(12))

BY (Expand ``lmatchp`` THEN HypCases ['12'] 3 ...)

1* 6. ∃s:Sub. s(1)=12 & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   >> ∃s:Sub. f(s(1))=f2(12) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(1))=f2(12))

2* 6. ∀s:Sub. ¬(s(1)=12)
   >> ∃s:Sub. f(s(1))=f2(12) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
      ∨ ∀s:Sub. ¬(f(s(1))=f2(12))
```

Figure 14: By 3 either 1 matches 12 or not.

```
* top 2 1 1 2 1 1
5. f=f2
6. ∃s:Sub. s(1)=12 & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
>> ∃s:Sub. f(s(1))=f2(12) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(1))=f2(12))

BY ((OnLast (SomeE ``s``) THEN ILeft THENM ITerm 's' ...) ...sc)
```

Figure 15: Let s match 1 with 12. Then s matches f(1) and f2(12).

```
* top 2 1 1 2 1 2
5. f=f2
6. ∀s:Sub. ¬(s(1)=12)
>> ∃s:Sub. f(s(1))=f2(12) & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   ∨ ∀s:Sub. ¬(f(s(1))=f2(12))

BY (IRight ...+s)

1* 5. s: Sub
   6. s(1)=12
   7. ∀s1:Sub. ¬(s1(1)=12)
   >> False
```

Figure 16: 1 does not match 12, so there is no matching substitution.

```
* top 2 1 1 2 1 2 1
5. s: Sub
6. s(l)=l2
7. ∀s1:Sub. ¬(s1(l)=l2)
8. f=f2
>> False

BY (EOn 's' 7 ...) THEN (Contradiction ...)
```

Figure 17: A match for f(l) and f2(l2) would match l and l2.

```
* top 2 1 1
1. l: Term list
2. t: Term
3. match?(t)
4. match?(l)
5. l2: Term list
>> ∃s:Sub. s(t).s(l)=l2 & mir(s) & (∀x:Var. x∈dom(s) <=> x∈t ∨ x∈l)
   ∨ ∀s:Sub. ¬(s(t).s(l)=l2)

BY (On 'l2' Unroll ...sc)

1* 6. t2: Term
   >> ∃s:Sub. s(t)=t2 & s(l)=l2 & min(s)
            & (∀x:Var. x∈dom(s) <=> x∈t ∨ x∈l)
      ∨ ∀s:Sub. ¬(s(t)=t2 & s(l)=l2)
```

Figure 18: In the induction step. s.

Figures 18 to 21 show some of the steps of lmatch_thm. To save space we have occasionally elided the conclusion of a subgoal when it is the same as the conclusion of the goal. The proof is by induction on the term list l. Figure 18 shows the first major step in the induction step. In this step, we are assuming that we know how to match t and l when possible and must show that t.l can be matched against an arbitrary l2 when possible. The step is to do a case analysis on whether l2 is empty or not. In Figure 19, we have an arbitrary t2 and l2 and must show that t.l can be matched against t2.l2 when possible. We do this by cases on whether t matches t2. If it doesn't, then no subsitution matches t.l to t2.l2. Otherwise, we proceed by cases on whether l matches l2 (Figure 20). If it does, then we proceed by cases on whether the substitutions s1 and s2 matching t to t2 and l to 2 respectively can be combined (Figure 21). If they can, then we have the desired substitution. If not (the first subgoal in Figure 21), then s1 and s2 are inconsistent and we can prove that there is no matching substitution by using hypotheses 8 and 12 and the two lemmas that characterize the application of a substitution to a term or term list by its values on variables that occur in the term or term list.

```
* top 2 1 1 1
3. match?(t)
4. match?(l)
5. l2: Term list
6. t2: Term
>> ∃s:Sub. s(t)=t2 & s(l)=l2 & min(s)
          & (∀x:Var. x∈dom(s) <=> x∈t ∨ x∈l)
   ∨ ∀s:Sub. ¬(s(t)=t2 & s(l)=l2)

BY (Expand ''matchp lmatchp'' THEN HypCases ['t2'] 3 ...)

1* 6. ∃s:Sub. s(t)=t2 & min(s) & ∀x:Var. x∈dom(s) <=> x∈t
   >> ...

2* 6. ∀s:Sub. ¬(s(t)=t2)
   >> ...
```

Figure 19: Does t match t2?.

```
* top 2 1 1 1 1
3. ∀l2:Term list. ∃s:Sub. s(l)=l2 & min(s) & (∀x:Var. x∈dom(s) <=> x∈l)
                ∨ ∀s:Sub. ¬(s(l)=l2)
6. ∃s:Sub. s(t)=t2 & min(s) & (∀x:Var. x∈dom(s) <=> x∈t)
>> ∃s:Sub. s(t)=t2 & s(l)=l2 & min(s) & (∀x:Var. x∈dom(s) <=> x∈t ∨ x∈l)
   ∨ ∀s:Sub. ¬(s(t)=t2 & s(l)=l2)

BY (HypCases ['l2'] 3 ...)


1* 6. ∃s:Sub. s(l)=l2 & min(s) & ∀x:Var. x∈dom(s) <=> x∈l
   >> ...

2* 6. ∀s:Sub. ¬(s(l)=l2)
   >> ...
```

Figure 20: Does l match l2?.

```
* top 2 1 1 1 1 1
5. ∃s:Sub. s(t)=t2 & min(s) & (∀x:Var. x∈dom(s) <=> x∈t)
6. ∃s:Sub. s(l)=l2 & min(s) & (∀x:Var. x∈dom(s) <=> x∈l)
>> ∃s:Sub. s(t)=t2 & s(l)=l2 & min(s) & (∀x:Var. x∈dom(s) <=> x∈t ∨ x∈l)
   ∨ ∀s:Sub. ¬(s(t)=t2 & s(l)=l2)

BY (SomeE ''s2'' 6 THEN SomeE ''s1'' 5 THEN
    CaseLemma 'sub_union' ['s1';'s2']...)

1* 5. s2: Sub
   6. s2(l)=l2
   7. min(s2)
   8. ∀x:Var. x∈dom(s2) <=> x∈l
   9. s1: Sub
   10. s1(t)=t2
   11. min(s1)
   12. ∀x:Var. x∈dom(s1) <=> x∈t
   13. ncst(s1,s2)
   >> ...

2* 5. s2: Sub
   6. s2(l)=l2
   7. min(s2)
   8. ∀x:Var. x∈dom(s2) <=> x∈l
   9. s1: Sub
   10. s1(t)=t2
   11. min(s1)
   12. ∀x:Var. x∈dom(s1) <=> x∈t
   13. ∃s:Sub. min(s) & s1⊂s & s2⊂s
              & ∀x:Var. x∈dom(s) => x∈dom(s1) ∨ x∈dom(s2)
   >> ∃s:Sub. s(t)=t2 & s(l)=l2 & min(s) & (∀x:Var. x∈dom(s))
```

Figure 21: Can the substitutions be combined?.

3.3 Applying Formal Metamathematics

How can these results be applied to Nuprl itself? One method is to define the type of terms to match exactly the terms of Nuprl. Then the theorems such as the match theorem apply to Nuprl itself. In this form they are *enlightening* but not *useful*. To make them useful it must be possible to apply them in proving theorems.

One way to do this is outlined in the second author's thesis [17]. The idea there is to directly connect terms with the objects they denote. He defines a type of term representatives similar to the type *Term* defined in this paper and a function that maps a term representative to the object it represents. This function is called *val* because it can be thought of as mapping a term representative to its *value*.

To see how the function *val* allows us to make the connection we want, consider the example of constructing term rewriting procedures in Nuprl (that is procedures mapping terms to terms that preserve equality). If we construct a function f in Nuprl of type $Term \rightarrow Term$ and prove that it respects *values*, so that $val(t) = val(f(t))$ for all members t of *Term*, then we can prove theorems in Nuprl using f as term-rewriting function. In the course of a proof, if we want to use f on some term a appearing in the goal, we first determine a representative t in *Term* for a. We then evaluate $f(t)$, getting a new member t' of *Term*. Because of the property known of f, we know that $val(t')$ is equal a and so we can substitute it for a. These steps have in effect rewritten a.

This approach is called *partial reflection*. In [17] is an extensive implementation of this approach. The partial reflection mechanism is developed completely within Nuprl, and several substantial applications are made. Note that matching is central to term rewriting, since the basic operation is applying a rewrite rule. A rewrite rule, in terms of our type *Term*, is a pair $\langle t, t' \rangle$ of terms containing variables, where all the variables of t' occur in t. A term u can be rewritten to a term u' via the rule $\langle t, t' \rangle$ if there is a substitution s for the variables of t such that u' is obtained by replacing an occurrence of $s(t)$ in u by $s(t')$. Thus to determine whether the rewrite rule applies to a term u, t must be matched against subterms of u. The development in [17] contains a substantial term rewriting system; this includes a formal treatment of matching similar to the one we developed for this paper, although the term structure there is somewhat different.

Another method of proceeding is to formalize the metatheory of Nuprl, making the types of terms and proof two of the basic types. The work here in proving metatheorems is about the same as in the partial reflection setting, and proofs in the formal metatheory, call it $Nuprl_1$, are just as enlightening. But now the connection to the base theory, call it $Nuprl_0$, is built-in. The algorithms extracted from a metatheorem about matching can be directly applied because they are about $Nuprl_0$ terms. Some of the details of this approach are presented in [20,19].

Acknowledgements

We would like to thank David Basin and Wilfred Chen for their helpful suggestions, and Elizabeth Maxwell for her help in preparing this document.

References

[1] P. Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77*. Amsterdam:North-Holland, 1978.

[2] W. Bledsoe and D. Loveland. *Automated Theorem Proving: After 25 Years*. American Math Soc., 1984.

[3] N. Bourbaki. *Theory of Sets*, volume I of *Elements of Mathematics*. Addison–Wesley, Reading, MA, 1968.

[4] R. Boyer and J. Moore. *A Computational Logic*. NY:Academic Press, 1979.

[5] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science.*, pages 103–84. NY:Academic Press, 1981.

[6] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. NY:Academic Press, 1983.

[7] A. Bundy. A broader interpretation of logic in logic programming. In *Proc. 5th Sympo. on Logic Programming*, 1988.

[8] R. L. Constable *et al. Implementing Mathematics with the Nuprl Development System*. NJ:Prentice-Hall, 1986.

[9] M. Davis. The prehistory and early history of automated deduction. In *Automation of Reasoning 1*, pages 1–28. Springer-Verlag, NY, 1983.

[10] M. Davis and J. T. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. *Comp. Math. with Applications*, 5:217–230, 1979.

[11] J. Gallier. Logic for computer science. In *Foundations of Automatic Theorem Proving*. Harper and Row, 1986.

[12] J.-Y. Girard. *Proof Theory and Logical Complexity, vol. 1*. Bibliopolis, Napoli, 1987.

[13] D. Good. Mechanical proofs about computer programs. In C. Hoare and J. Shepardson, editors, *Mathematical Logic and Programming Languages*, pages 55–75. Springer-Verlag, NY, 1985.

[14] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *Lecture Notes in Computer Science*, 78, 1979.

[15] D. M. Harper, R. and R. Milner. Standard ml. Technical report, Lab. for Foundations of Computer Science, University of Edinburgh, 1986. TR ECS-LFCS-86-2.

[16] S. Hiyashi and H. Nakano. *PX: A Computational Logic*. Foundations of Computing. MIT Press, Cambridge, MA, 1988.

[17] D. J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[18] D. J. Howe. Computational metatheory in Nuprl. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 238–257, New York, 1988. Springer-Verlag.

[19] T. Knoblock. *Mathematical Extensibility in Type Theory.* PhD thesis, Cornell University, 1987.

[20] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Annual Symp. on Logic in Computer Science.* IEEE., 1986.

[21] P. A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, page 27, January 1988.

[22] P. Martin-Lof. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. Amsterdam:North Holland, 1982.

[23] P. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, Ithaca, NY, 1988.

[24] Y. Moschovaski. *Elementary Induction on Abstract Structures.* North-Holland, Amsterdam, 1974.

[25] K. Mulmuley. The mechanization of existence proofs of recursive predicates. In *Seventh Conf. on Automated Deduction, LNCS 170*, pages 460–475. Springer-Verlag, NY, 1984.

[26] J. S. Newell, A. and H. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proc. West Joint Computer Conf.*, pages 218–239, 1957.

[27] L. Paulson. Lessons learned from LCF: a survey of natural deduction proofs. *Comp. J.*, 28(5), 1985.

[28] L. Paulson. *Logic and Computation.* Cambridge University Press, NY, 1987.

[29] P. Rosenblum, J. Laird, and A. Newell. Metalevels in soar. In D. N. P. Maes, editor, *Meta-Level Architectures and Reflection.* North-Holland, 1988.

[30] N. Shanker. Towards mechanical metamathematics. *J. Automated Reasoning*, 1(4):407–434, 1985.

[31] R. M. Smullyan. *First-Order Logic.* Springer–Verlag, New York, 1968.

[32] H. Wang. Toward mechanical mathematics. *IBM J. Research and Development*, (4):2–22., 1960.

[33] L. Wos, R. Overbeek, L. Ewing, and J. Boyle. *Automated Reasoning.* Prentice–Hall, Englewood Cliffs, NJ, 1984.

Computational Metatheory in Nuprl *

Douglas J. Howe
Department of Computer Science
Cornell University

February 29, 1988

Abstract

This paper describes an implementation within Nuprl of mechanisms that support the use of Nuprl's type theory as a language for constructing theorem-proving procedures. The main component of the implementation is a large library of definitions, theorems and proofs. This library may be regarded as the beginning of a book of formal mathematics; it contains the formal development and explanation of a useful subset of Nuprl's metatheory, and of a mechanism for translating results established about this embedded metatheory to the object level. Nuprl's rich type theory, besides permitting the internal development of this partial reflection mechanism, allows us to make abstractions that drastically reduce the burden of establishing the correctness of new theorem-proving procedures. Our library includes a formally verified term-rewriting system.

1 Introduction

Most theorem-proving systems that allow the user to soundly extend the inference mechanism with new theorem-proving procedures use one of two

*This research was supported in part by NSF grant CCR-8616552. This paper is to be published in *The Proceedings of the Ninth International Conference on Automated Deduction* (in the *Lecture Notes in Computer Science* series), Springer-Verlag, May 1988.

approaches. The first approach is to require that new procedures be proven correct in a formalized metatheory [5,2,14,11]. The second is to provide a tactic mechanism [7,4], which permits arbitrary new procedures, but which requires each application of such a procedure to generate a proof in terms of the primitive inference rules. It appears that only the second approach has been used in significant applications. In this paper we show how Nuprl [4], which incorporates a tactic mechanism, can be "boot-strapped" to encompass both approaches. We show how Nuprl's powerful type theory can be used to develop a completely internal account of a portion of its metatheory, and to make abstractions that significantly reduce the burden of formally verifying new theorem proving procedures. This partial reflection mechanism, together with some applications, has been implemented in Nuprl, resulting in a completely formal account of an extension to Nuprl that allows new theorem-proving procedures to be programmed in the type theory. This implementation provides some evidence for the practical potential of the approach.

As a simple example of the limitations of the tactic mechanism, consider equations involving an associative commutative operator "$\cdot$". One way to check that an equation

$$a_1 \cdot a_2 \cdot \ldots \cdot a_m = b_1 \cdot b_2 \cdot \ldots \cdot b_n$$

holds is to sort the sequences $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$, with respect to some ordering on terms, and check that the results are identical. A tactic that emulated this informal procedure would have to chain together appropriate instances of lemmas (*e.g.*, for the associativity and commutativity of $\cdot$) and rules (*e.g.*, the substitution rule). This indicates two major problems. First, the tactic writer must be continually concerned with generating Nuprl proofs, and this can increase the intellectual effort involved in constructing theorem-proving procedures. Secondly, there is a major efficiency problem. In the example, even though it is known in advance that the sorting algorithm is sufficient to establish equality, every time the tactic is called it must "re-justify" the algorithm.

The mechanisms we have implemented allow procedures such as the one just described to be used directly. The main part of our implementation consists of a large collection of Nuprl definitions, theorems and proofs. This *library* may be regarded as the beginning of a book of formal mathematics;

it contains the formal development and explanation of a useful subset of Nuprl's meta-theory, and of a mechanism for translating results established about this embedded meta-theory to the object level. The most important application of this is to the automation of reasoning: one can write Nuprl programs that directly encode such meta-level procedures as the sorting-based algorithm given above, prove that the program is correct, and then apply it in the construction of Nuprl proofs. Applications of such programs can be done in a manner consistent with Nuprl's proof development paradigms.

The core of our library contains the development of the partial reflection mechanism. Natural representations for a certain subclass of the terms of the type theory (roughly, the quantifier-free terms) and for contexts (*i.e.*, definitions and hypotheses) are constructed. Theorems are then proven in Nuprl that connect the representations to the objects represented. This allows the results of computations involving representations to be used in making judgments about the represented objects. The remainder of the library contains the development of several applications. These include a term rewriting system and a procedure involving the algorithm discussed above.

The basic idea of the reflection mechanism is simple. Using Nuprl's recursive-type constructor, we define a type `Term0`[1] that represents a certain subset of the terms of Nuprl's theory. We then define what it means for a term (i.e., a member of `Term0`) to be well-formed with respect to an environment, which is a Nuprl object which associates atoms with Nuprl objects. Environments are used to represent definitions and hypotheses. Let `Term`(α) be the type of all members of `Term0` that are well-formed with respect to α. We can demonstrate the existence of Nuprl functions `type` and `val` such that for any α,

$$\texttt{type}(\alpha) \quad \texttt{in} \quad \texttt{Term}(\alpha)\texttt{->SET}$$

and

$$\texttt{val}(\alpha) \quad \texttt{in} \quad \texttt{t:Term}(\alpha) \texttt{ -> type}(\alpha)\texttt{(t)},$$

where a member of SET is a type together with an equality relation on that type.

[1] We will use typewriter typeface for terms (that possibly contain definition instances) of Nuprl's type theory. Italicized identifiers within these terms will be meta-variables.

To see how the above can be put to use, consider a simplified example. Suppose that we have constructed a function f of type

$$\texttt{Term0 -> Term0},$$

and that we can show that it preserves equality in some environment α, *i.e.*, that for any term t in $\texttt{Term}(\alpha)$,

$$\texttt{type}(\alpha)(t) = \texttt{type}(\alpha)(f(t)) \texttt{ in SET}$$

and

$$\texttt{val}(\alpha)(t) = \texttt{val}(\alpha)(f(t)) \texttt{ in type}(\alpha)(t)$$

(*i.e.*, for the second equality, the equivalence relation from $\texttt{type}(\alpha)(t)$ is satisfied). For example, f might be based on the sorting procedure mentioned earlier. Suppose that we are in the course of proving a Nuprl theorem, that the current goal is of the form >> T (where >> is Nuprl's turnstile), and that we want to "apply" f to T. The first step is to *lift* the goal. To do this, we apply a special tactic which takes as an argument the environment α. This tactic first computes a member t of $\texttt{Term}(\alpha)$ such that the application $\texttt{val}(\alpha)(t)$ computes to a term identical to T, and then it generates the subgoal

$$\texttt{>> val}(\alpha)(t)$$

(assuming that the environment α is appropriate). We now apply a tactic which, given f, has the effect of computing $f(t)$ as far as possible, obtaining a value $t' \in \texttt{Term}(\alpha)$, and producing a subgoal

$$\texttt{>> val}(\alpha)(t').$$

At this point, we can proceed by applying other rewriting functions, or by simply computing the conclusion and obtaining an "unlifted" goal >> T' to which we can apply other tactics.

An important aspect to this work is that the partial reflection mechanism involves no extensions to the logic; the connection between the embedded meta-theory and the object theory is established completely within Nuprl. This means, first, that the soundness of the mechanism has been

formally verified. Secondly, there is considerable flexibility in the use of the mechanism. All of its components are present in the Nuprl library, so one can use them for new kinds of applications without having to do any metatheoretic justification. Rewriting functions are just an example of what can be done; many other procedures that have proven useful in theorem proving, congruence closure for example, can be soundly added to Nuprl. Thirdly, the mechanism provides a basis for stating new kinds of theorems. For example, one can formalize in a straightforward way the statement of a familiar theorem of analysis: *if $f(x)$ is built only from x, $+$, $\cdot$, ..., then f is continuous.*

Finally, and perhaps most importantly for the practicability of this approach, it is possible to make abstractions that drastically reduce the burden of verifying the correctness of new procedures. Since contexts (environments) are represented, one can prove the correctness of a procedure once for a whole class of applications. For example, a simplification procedure for rings could be proven correct for any environment where the values of certain atoms satisfy the ring axioms, and then be immediately applied to any particular context involving a ring. A general procedure such as congruence closure could be proved once and for all to be correct in any environment whatsoever. Another kind of abstraction is indicated by the rewriting example to be given later. Functions such as **Repeat** are proved to map rewriting functions to rewriting functions; these kinds of combinators can often reduce the proof of correctness for a new rewriting function to a simple typechecking task that can be dealt with automatically.

One of the main motivations of this work is to provide tools that will be of use in formalizing Bishop-style constructive real analysis [1]. The design of the reflection mechanism incorporates some of the notions of Bishop's set theory. However, our work is more generally applicable, since Nuprl's type theory is capable of directly expressing problems from many different areas. It should be of use in other areas of computational mathematics (e.g., in correct-program development). Using the techniques developed in AUTOMATH [6] and LF [8], it can be applied to reasoning in a wide variety of other logics (constructive in character or not). For example, any first-order logic can be embedded within Nuprl; in such a case, the reflection mechanism will encompass the quantifier-free portion of the logic.

In the next section is a brief description of some of the relevant compo-

nents of Nuprl. This is followed by a concrete example of term-rewriting. The following section gives some details about what has been implemented. Finally, there is a discussion of some related work, and some concluding remarks are made. For a more complete account of the work described in this paper, and for a complete listing of the library that was constructed, see [9].

2 Nuprl

Nuprl [4] is a system that has been developed at Cornell by a team of researchers, and is intended to provide an environment for the solution of formal problems, especially those where computational aspects are important. One of the main problem-solving paradigms that the system supports is that of *proofs-as-programs*, where a program specification takes the form of a mathematical proposition implicitly asserting the existence of the desired programs. Such a proposition can be formally proved, with computer assistance, in a manner resembling conventional mathematical arguments. The system can extract from the proof the implicit computational content, which is a program that is guaranteed to meet its specification. With this paradigm, the construction of correct programs can be viewed as an applied branch of formal constructive mathematics.

The logical basis of Nuprl is a constructive type theory that is a descendent of a type theory of Martin-Löf [12]. The rules of Nuprl deal with *sequents*, which are objects of the form

$$x_1 : H_1,\ x_2 : H_2,\ \ldots,\ x_n : H_n \gg A.$$

Sequents, in the context of a proof, are also called *goals*. The terms H_i are referred to as *hypotheses* or *assumptions*, and A is called the *conclusion* of the sequent. Such a sequent is said to be true when, roughly, there is a procedure that, under the assumption that H_i is a type for $1 \leq i \leq n$, takes members x_i of the types H_i, and produces a member of the type A. An important point about the Nuprl rules is that they allow top-down construction of this procedure. They allow one to *refine* a goal, obtaining subgoal sequents such that a procedure for the goal can be computed from procedures for the subgoals.

The Nuprl type theory has a large set of type constructors. They can be roughly described as follows. The type `Int` is the type of all integers. `Atom` is the type of all character strings (delimited by double quotes). The disjoint union $A \mid B$ has members `inl(`a`)` and `inr(`b`)` for $a \in A$ and $b \in B$. The dependent product $x : A \# B$ (written $A \# B$ when x does not occur free in B) has members $\langle a, b \rangle$ for $a \in A$ and $b \in B[a/x]$. The dependent function space $x : A \text{->} B$ (written $A \text{->} B$ when x does not occur free in B) has as members all functions $\lambda x.b$ such that for all $a \in A$, $b[a/x] \in B[a/x]$. The type A `list` has as members all lists whose elements are in A. The type $\{x : A \mid B\}$ is the collection of all members of A such that $B[a/x]$ has a member. The type a=b `in` A (for $a, b \in A$) has the single member `axiom` if a and b are equal as members of A, and has no members otherwise. The members of the recursive type `rec(`T`.`A`)` are the members of $A[\texttt{rec}(T.A)/T]$. Finally, there is a cumulative hierarchy of universes `U1`, `U2`, ..., where each `Ui` is obtained via closing under the other type constructors.

Associated with each type constructor are forms for making use of members of the type; *e.g.*, projection for pairs, an induction form for integers, and application for functions. These forms give rise to a system of computation, and the proceeding discussion of membership, to be accurate, must be amended to reflect the fact that $b \in A$ whenever $a \in A$ and b computes to a. The usual connectives and quantifiers of predicate calculus can be defined in terms of the above type constructors using the *propositions-as-types* correspondence.

Proofs in Nuprl are tree-structured objects where each node has associated with it a sequent and a refinement rule. The children of a node are the subgoals (with their subproofs) which result from the application of the refinement rule of the node to the sequent. A refinement rule can be either a primitive inference rule, or a tactic (written in the ML language [7]). If it is a tactic, then there is a hidden proof tree whose leaves are the children of the refinement.

Nuprl has a large number of inference rules; we only point out one which is of special significance for our work. The *evaluation rule* allows one to invoke Nuprl's (reasonably efficient) evaluator on either a hypothesis or the conclusion of a sequent. The term is given as input to the evaluator, is computed as far as possible, and is replaced in the sequent by the result of the computation. This rule is used to execute theorem-proving procedures

that are based on the reflection mechanism.

Interactions with Nuprl are centred on the *library*. A library contains an ordered collection of definitions, theorems, and other objects. New objects are constructed using special-purpose window-oriented editors. The text editor, together with the definition facility, permit very readable notations for objects of Nuprl's type theory. The proof editor is used to construct proofs in a top-down manner, and to view previously constructed proofs. Proofs are retained by the system, and can be later referred to either by the user, or by tactics.

3 An Example

We will now look at an example developed in our library, where a simple rewriting function is applied to an equation involving addition and negation over the rational numbers. The environment used is α_Q, which contains "bindings" for rational arithmetic. For example, it associates to the Nuprl atom "Q" the set (*i.e.*, member of Set) Q of rational numbers, and to the atom "Q_plus" the addition function. The rewriting functions for this environment are collected into the type Rewrite(α_Q). The requirement for being a rewriting function over α_Q is actually stronger than indicated earlier; the criteria given there must hold for any α which extends α_Q.

The member of Rewrite(α_Q) we will apply is

```
Repeat( TopDown( rewrite[x;y]( -(x+y) -> -x+-y ) ) )
```

(this function is named norm_wrt_Q_neg). This does just what one would expect: given a term, it repeatedly, in top-down passes, rewrites subterms of the form -(x+y) to -x+-y. The functions Repeat and TopDown both have type

$$\texttt{Rewrite}(\alpha) \texttt{ -> } \texttt{Rewrite}(\alpha)$$

for any α. The argument to TopDown is by definition the application of a function of three arguments, which must be: a list of atoms that indicate what constants are to be interpreted as variables for the purpose of matching; and two members of Term0 which form the left and right sides of a rewrite-rule. Nuprl definitions are used to make members of Term0 appear

as close as possible to the terms they represent. The term x+y that appears in the definition of norm_wrt_Q_neg is, when all definitions are expanded,

```
inl( < "Q_plus", <inl(<"x",nil>) . inl(<"y",nil>) . nil > ).
```

The sequent we apply our function to is

```
>> -(w+x+y*z+z) = -w+-x+-(y*z)+-z in Q
```

(the hypotheses declaring the variables to be rationals have been, and will be, omitted). The first step is to apply the tactic

```
(LiftUsing ['α_Q'] ...),
```

(where the "..." indicates the use of a Nuprl definition that applies a general purpose tactic called the *autotactic*) which generates the single subgoal

```
α:Env, (...)
>> ↓( val( α, -(w+x+y*z+z) = -w+-x+-y*z+-z in Q ) )
```

(↓ is an information-hiding operator that will not be discussed here—see [4] for an explanation). The notation "(...)" indicates that the display of a hypothesis has been suppressed by the system. This elided hypothesis contains (among other information) the fact that α is equal to an environment formed by extending α_Q by entries for representatives of the variables w, x, y and z. The next step is to apply the tactic

```
(RewriteConcl 'norm_wrt_Q_neg' ...),
```

which generates a single subgoal with conclusion

```
↓( val( α,-w+-x+-(y*z)+-z = -w+-x+-(y*z)+-z in Q ) ).
```

The proof is completed using an equality procedure to be described later.

4 The Implementation

In this section is a presentation of the Nuprl library containing the partial reflection mechanism and applications. The library contains more than

1300 objects, and is by far the largest development undertaken so far using Nuprl. An ideal description of the library would involve the use of the Nuprl system itself; the system is designed not only for the construction of formal arguments, but also for their presentation. The best that can be done here is to explain some of the basic definitions and theorems. The next three subsections are devoted to the library objects and tactics that establish the reflection mechanism. The last two subsections describe some implemented examples of the use of the mechanism.

Of the 1300 objects in the library, about 800 are theorems, 400 are definitions, and 100 are ML objects. The library can be roughly divided according to topic (in order) as follows.

- Basics.
- Lists.
- Sets.
- The representation of Nuprl terms.
- Association lists and type environments.
- Function environments and combined environments.
- Booleans and "partial" booleans.
- Well-formedness and evaluation of terms.
- The monotonicity of certain functions with respect to environment extension, and operations on environments.
- Term rewriting.
- Other applications.

The library divisions corresponding to the first three topics are of somewhat lesser interest, and will be discussed very briefly in this section.

Before proceeding, we need to establish a few notational conventions. Typewriter typeface will be used exclusively to denote Nuprl terms or names of Nuprl objects. In the context of a Nuprl term, a variable which

is in italics will denote a meta-variable. Thus `x+y` denotes a Nuprl term where `x` and `y` are *Nuprl* variables, whereas in $x+y$, x and y range over Nuprl terms. We will be somewhat sloppy in the use of meta-variables, using them, for example, to denote parameters of Nuprl definitions, but the context should make clear what the intention is.

The presentation below is somewhat in the style of conventional mathematics. Usually, explicit references to the existence of Nuprl objects will not be made. Unless stated otherwise, when a definition is made, there is a corresponding definition in the library; and when it is asserted that a certain statement, written in Nuprl syntax, is true, or can be proved, then there is a corresponding theorem in the library.

The "basics" section of the library contains mostly simple definitions, such as for logical concepts and definitions pertaining to the integers. There are also definitions that provide alternate notations for simple combinations of base Nuprl terms. For example, we define a "let" construct in the usual way:

$$\texttt{let } x = t \texttt{ in } t' \equiv (\lambda x.t')(t),$$

and use it (together with the definitions for projection from pairs) in the definition `let2`:

$$\texttt{let } x,y = p \texttt{ in } t \equiv \texttt{let } x = p.1 \texttt{ in let } y = p.2 \texttt{ in } t.$$

The "lists" section of the library contains a fairly substantial development of list theory.

In order to account for equivalence relations that are not built into the type theory[2] (*e.g.*, equality of real numbers), we define a *set* to be a pair consisting of a type together with an equivalence relation on the type. We define a hierarchy of collections of sets that parallels the universe hierarchy: a member of `Set(i)` is a pair consisting of a type from `Ui` together with an equivalence relation on that type. For S a set, define $|S|$ to be the type component of S, and for a and b members of $|S|$, define $a=b$ `in` S to be the application of the equivalence relation of S to a and b. We also define some particular sets that will be used later in the library: `Set` and `SET` are `Set(6)` and `Set(7)`, respectively, and

```
Prop  ≡  <U6, λ P,Q. P <=> Q>.
```

[2] Nuprl has a quotient-type constructor, but it cannot be used for our purposes (see [4])

The choices of 6 and 7 are somewhat arbitrary; we only need to guarantee that the levels are high enough to accommodate anticipated applications.

It is a simple matter to define some basic set constructors. We will overload our notations and use S_1->S_2 to denote the *set* of functions from S_1 to S_2, and S_1#S_2 for the product of S_1 and S_2. Also, #(L) will denote the product of the sets in the list L. We will not discuss here our treatment of Bishop's notion of *subset* [1].

4.1 Representation

In this section we discuss first the representation in Nuprl of a simple class of Nuprl terms, and then the representation of contexts (*i.e.*, the library and hypotheses).

The type **Term0** of "raw" terms, which do not necessarily represent an object of the Nuprl theory, is defined via the recursive type

```
rec(T. (Atom#T list) | (T#T#Atom) | (Atom#T#T) | Int ).
```

Members of this type, which will be referred to as *meta-terms*, or just *terms* when no confusion can result, can be thought of as trees with four kinds of nodes. The meaning of the different kinds of nodes is suggested by the notations for the injections:

$f(l)$ ≡ inl(<f,l>)
x = y in A ≡ inr(inl(<x,y,A>))
y{i x} ≡ inr(inr(inl(<i,x,y>)))
n ≡ inr(inr(inr(n))).

The above are all members of **Term0** whenever f, A, i are in **Atom**, x and y are in **Term0**, l is a list of members of **Term0**, and n is an integer. These kinds of nodes will be referred to, respectively, as function-application nodes, equality-nodes, i-nodes, and integer nodes. The first will represent terms which are function applications; the second, terms which are equalities (in the set sense); and the last, terms which are integers. The inclusion of i-nodes is motivated by Bishop's notion of *subset* [1]; i-nodes will be ignored in the rest of this paper.

Contexts are represented with association lists (or "a-lists"), of type

```
(Atom # A) list
```

for A a type. Such lists are used to associate Nuprl objects with the atoms that appear in meta-terms. In dealing with a-lists, and in the term-rewriting system described later, extensive use is made of *failure*. For A a type, define $?A$ to be the type $A|\texttt{True}$ (where `True` is a single-element type). A value $\texttt{inl}(a)$ of $?A$ is denoted by $\texttt{s}(a)$ (`s` for "success"), and the unique value of the form $\texttt{inr}(a)$ is denoted by `fail`. In later sections, the distinction between a and $\texttt{s}(a)$ will often be glossed. The library has many objects related to failure and a-lists.

A basic definition for a-lists is of a-list application, written $l\{A\}(a)$. If A is a type, if l is in $\texttt{Atom}\#A$ `list`, and if a is in `Atom`, then $l\{A\}(a)$ is in $?A$. Evaluation of $l\{A\}(a)$ either results in failure, or returns some b such that $\texttt{<}a,b\texttt{>}$ is a member of l.

Contexts are represented as pairs of a-lists, the first component of which is a type environment, and the second of which is a function environment. Type environments associate atoms with members of `Set` and are members of the type:

$$\texttt{TEnv} \equiv \texttt{(Atom \# S:Set \# ?triv\_eq(S)) list}.$$

The predicate `type_atom` is defined so that $\texttt{type\_atom}(\gamma,a)$ is true if and only if either a is the atom "`Prop`" or a is bound in the type environment γ. Also, $\texttt{type\_atom}(\gamma,a)$ has the property that if, during the course of a proof, γ and a are sufficiently concrete (*e.g.*, if they do not contain any free variables), it can be evaluated to either `True` or `False`. Define

$$\texttt{AtomicMType}(\gamma) \equiv \{\ \texttt{a: Atom} \mid \texttt{type\_atom}(\gamma,\texttt{a})\ \}$$

(`M` is for "meta"). For members a of this type, define $\gamma(a)$ to be $\texttt{<Prop, fail>}$ if a is "`Prop`", otherwise the value obtained from looking up a in γ. We define

$$\texttt{MType}(\gamma) \equiv \{\ \texttt{t: Atom list \# Atom} \mid \texttt{all\_type\_atoms}(\gamma,\texttt{t})\ \}.$$

This is the type of "meta-types" for functions; the first component of such a meta-type is a list of atoms that represent a function's argument types, and the second component represents the result type.

We can now define evaluation for members of $\texttt{MType}(\gamma)$. If mt is such a member, and $mt.1$ (*i.e.*, the first component of the pair mt) is non-empty,

then we can evaluate mt's "domain type":

```
dom_val(γ,mt)  ≡
   let l,b = mt in
   #((map λa. val(γ,a) on l to SET list)).
```

Evaluation of function meta-types is defined as:

```
val(γ,mt)  ≡
let l,b = mt in
if null(l) then val(γ,b) else dom_val(γ,mt) -> val(γ,b).
```

A value $\mathtt{val}(\alpha,z)$ will often be referred to as the *meaning* or *value* of z in α.

Function environments associate an atom, called a *meta-function*, with a meta-type, with a value that is a member of the value of the meta-type, and with some additional information about the value. More specifically, for γ a type environment, define

```
FEnv(γ)  ≡
Atom # mt:MType(γ) # f:|val(γ,mt)| # val_kind(γ,mt,f)  list.
```

The type $\mathtt{val\_kind}(\gamma,mt,f)$ can be ignored here. Definitions analogous to those made for type environments are made for function environments. Thus define function-environment application, the predicate `fun_atom`, and the type $\mathtt{MFun}(\alpha)$ of meta-functions. For f in $\mathtt{MFun}(\alpha)$, $\mathtt{mtype}(\alpha,f)$ is the meta-type assigned to f by $\alpha.2$, and $\mathtt{val}(\alpha,f)$ is the value assigned to f.

We can now define the type whose members represent contexts:

```
Env  ≡  γ:TEnv # FEnv(γ).
```

Members of `Env` will simply be called *environments*, and the variable α will always denote an environment. For most of the definitions presented so far that take a type environment as an argument, there is a new version which takes an environment as an argument. In what follows, only these new versions will be referred to, so the same notation will be used.

4.2 Well-Formedness and Evaluation

This section presents the core of the partial reflection mechanism. We formalize a notion of well-formedness for members of **Term0**, and construct an evaluation function for well-formed terms.

Roughly, a meta-term t is well-formed in an environment α if "the meta-types match up". For example, if t is a function-application $f(l)$, then we require that the *domain component* of the meta-type associated with f in α "matches" the list of terms l; *i.e.*, that they have the same length as lists, and that the atomic meta-types of f's domain "match" the atomic meta-types computed for the terms in l. A simple definition of "match" for atomic meta-types would be equality as atoms. However, since each f can have only one meta-type associated with it, this definition would preclude us from representing such simple terms as **x+y**, where **x** and **y** are Nuprl variables declared to be in a sub-type of the integers, and where **+** is addition over the integers. Also, it would preclude us from dealing with simple partial functions, *i.e.*, from representing terms like $g(x)$ where the domain type of g is a subtype of the type of x and x satisfies the predicate of the subtype. This leads us to consider a notion of matching of atomic meta-types that extends the simple version by allowing the associated values to stand in a subtype relationship. Consider the case of a meta-term $f(t)$, where f has meta-type domain the singleton list $[A]$, t has the meta-type B, and the value in α of A is a subtype of the value of B. Then for $f(t)$ to be well-formed we require that the *value* of t satisfy the subtype predicate associated with A.

Thus well-formedness depends on evaluation, which in turn is only defined on well-formed terms, and so we need to deal with these notions in a mutually recursive fashion. In particular, we first define the function which computes meta-types of terms, the evaluation function, and the well-formedness predicate, and then prove that they are well-defined (*i.e.*, have the appropriate types) simultaneously by induction.

Several functions below are defined by recursion. We will give the recursive definitions informally, although the formal versions are obtained directly.

The meta-type of a well-formed term is simple to compute; define **mtype(**α**,**t**)** to be: if t is $f(l)$, then **mtype(**α**,**f**).2**; if t is **u=v in** A,

then "Prop"; if t is n then "Int". Also, define type(α,t) to be the value in α of t's metatype. Evaluation of terms is defined using an auxiliary function that applies a function g to a list l and produces a tuple (denoted by $g\{\alpha\}(l)$) of the results. Thus we define val(α,t) to be: if t is $f(l)$, then

```
if null(mtype(α,f).1) then val(α,f)
else val(α,f) (g{α}(l))
```

where $g(t)$ is val(α,t); if t is u=v in A, then

```
val(α,u)=val(α,v) in val(α,A);
```

and if t is n, then n.

Due to space limitations, we will not give a detailed definition of the *well-formedness predicate*. Instead, we will just informally describe the criteria for a function-application $f(l)$ to be well-formed in an environment α. First, f must be bound as a function in α, and the domain part of its meta-type must be a list r with the same length as l. Secondly, the terms in l must be well-formed in α. *Finally, for each t in l and corresponding a* in r, a and mtype(α,t) must "match" in the sense outlined earlier in this section. Note that this last criterion involves val(α,t).

The assertion that a term t is well-formed in an environment α is written wf@(α,t). The central theorem of the partial reflection mechanism is the following:

```
∀α:Env. ∀t:Term0. wf@(α,t) in U &
↓(wf@(α,t)) => (mtype(α,t) in AtomicMType(α)
                & val(α,t) in |type(α,t)|)
```

(where U is a particular universe—we use membership in a universe to assert that a term is a well-formed type). Finally, define Term(α) to be the set of all meta-terms that are well-formed in α.

For the reflection method to be useable, it is necessary that the lifting procedure outlined in the first section of this chapter be reasonably fast. Since each lifting of a sequent will require establishing that some meta-terms are well-formed, the well-formedness predicate wf@ is not suitable, imposing excessive proof obligations. Instead, we use a characterization wf of the predicate that embodies an algorithm for the "decidable part" of

well-formedness. To prove $\mathtt{wf}(\alpha, t)$ for concrete α and t, we apply Nuprl's evaluation rule, which simplifies the formula to a conjunction of propositions that assert that some terms satisfy some subtype predicates. In many cases arising in practice, the simplified formula will be `True` (or `False`).

The characterization `wf` is accomplished using the "partial booleans". Define

$$\mathtt{Bool} \equiv \mathtt{True} \vee \mathtt{True} \quad \text{and} \quad \mathtt{PBool} \equiv \mathtt{Bool} \vee \mathtt{U},$$

so a partial boolean is either a boolean or a proposition. We define `PBool` analogues of the propositional connectives, and of some of the other basic predicates. The characterization is built essentially by replacing components of `wf@` by their `PBool` analogues.

We will usually want our assertions about the correctness of theorem-proving procedures to respect environment extension. For example, we may have constructed a function f that normalizes certain expressions and that is correct in an environment α_0 that associates certain atoms with the operations of rational arithmetic. Clearly we will want to do more than just apply f to terms that are well-formed in α_0; at the very least, we will want to apply f to a term like `x+y`, where `x` and `y` are variables declared in some sequent. This necessitates a notion of *sub-environment*. The environment α_1 is a sub-environment of α_2, written $\alpha_1 \subset \alpha_2$, if $\alpha_1.1$ and $\alpha_1.2$ are sub-a-lists of $\alpha_2.1$ and $\alpha_2.2$, respectively. Most of the functions that have been defined so far that take an environment as an argument are *monotonic* with respect to environment extension. For example, if $\alpha_1 \subset \alpha_2$, and if t is well-formed in α_1, then t is well-formed in α_2 and its values in the two environments are equal. A substantial portion of the library is devoted to monotonicity theorems.

4.3 Lifting

Suppose that we wish to use some procedure that operates on meta-terms to prove a sequent

$$A_1,\ A_2,\ \ldots,\ A_n \gg A_{n+1}.$$

We first *lift* the sequent using the tactic invocation `LiftUsing` *envs*, where *envs* is an ML list of Nuprl terms that are environments, call them $\alpha_1, \alpha_2,$

$\ldots, \alpha_m$. All but one of the subgoals generated by this tactic will in general be proved by the autotactic (a general purpose tactic that runs after all other top-level tactic invocations). The remaining subgoal will have the form

$$\alpha{:}\texttt{Env},\ (\ldots),\ \texttt{val}(\alpha, A'_1),\ \texttt{val}(\alpha, A'_2),\ \ldots,\ \texttt{val}(\alpha, A'_n)$$
$$>>\ \texttt{val}(\alpha, A'_{n+1}),$$

where $(\ldots)$ is an elided hypothesis, and each A'_i is a member of Term0 that represents A_i. By the use of definitions, A'_i will usually appear to the user to be exactly the same as A_i. The preceding description of lifting contains two simplifications. First, the hypotheses shown may be interspersed with hypotheses that have declared variables; the lifting procedure ignores such hypotheses. Secondly, some of the A_i may remain in the hypothesis list, not being replaced by $\texttt{val}(\alpha, A'_i)$. This happens when the representation computed for A_i is trivial.

The first step in lifting a sequent is to apply a procedure that uses the α_j's to compute the representations A'_i in Term0 for the terms A_i, $1 \leq i \leq n+1$. This procedure also produces a list of environment additions for those subterms that are unanalyzable with respect to the α_j's; each such subterm s will be represented by a new metafunction constant whose associated value will be s. The new variable α is added to the sequent to be lifted, as well as the new hypothesis $(\ldots)$. This elided hypothesis contains the following assertions: that the α_j are consistent (*i.e.*, that they agree on the intersection of their domains); that α is equal to the combination (call it $\overline{\alpha}$) of the α_j and the environment additions; that $\alpha_j \subset \alpha$ for each j; and that each A'_i is a well-formed meta-term that has meta-type "Prop". Of course, to justify the addition of this hypothesis, the lifting tactic must prove each of the assertions. This is accomplished by the use of simplification (using the evaluation rules) and of lemmas (to handle some details concerning the fact that some substitutions of $\overline{\alpha}$ for the variable α must be done before simplification). Most of these assertions will be proved automatically. The final stage of the lifting procedure is to replace each A_i by $\texttt{val}(\alpha, A'_i)$; the tactic only needs to use evaluation in order to justify this.

The elided hypothesis contains most of the information necessary to prove the applicability of meta-procedures. For example, if f is in

$$\texttt{Rewrite}(\alpha')$$

then to prove that f is applicable all that needs to be shown is that $\alpha' \leq \alpha$. An important point about this new hypothesis is that it is easy to maintain. For example, rewriting functions preserve meta-types and values, so when a rewrite is applied to a sequent, we can quickly update the hypothesis to reflect the changes in the sequent. Thus, the work of applying a rewrite to a lifted sequent is dominated by the work of doing the actual rewriting.

4.4 Term Rewriting

The most important application of the reflection mechanism is to equality reasoning. In this section is a description of the implementation in Nuprl of a verified term-rewriting system. This implementation consists of a collection of definitions, theorems, and tactics that aid the construction of term-rewriting (i.e., equality-preserving) functions. Using this collection, one can easily turn equational lemmas into rewrite rules, and concisely express a wide variety of rewriting strategies. Proving the correctness of rewriting strategies generally reduces to simple typechecking.

4.4.1 The Type of Rewriting Functions

Informally, a function f of type `Term0->?Term0` is a rewrite with respect to an environment α if for any term t that is well-formed in some extension α' of α, $f(t)$ either fails or computes to a term t' such that: (1) t' is well-formed in α'; (2) t' and t have the same meta type, call it A; and (3) the meanings of t and t' are equal in the set which is the meaning of A. Define `Rewrite`(α) to be the collection of all such f. Note that monotonicity (with respect to environment extension) is built into the definition of `Rewrite`, so that if f is in `Rewrite`(α), and $\alpha \leq \alpha'$, then f is in `Rewrite`(α').

4.4.2 Basic Rewriting Functions

The basic units of conventional term rewriting are *rewrite rules*. These are ordered pairs $\langle t, t' \rangle$ of terms containing variables, where the variables of t' all occur in t. A term u rewrites to a term u' via the rule $\langle t, t' \rangle$ if there is a substitution σ for the free variables of t such that u' is obtained by replacing an occurrence of $\sigma(t)$ in u by $\sigma(t')$. In our setting, we will view a rewrite rule as a member of `Rewrite`(α), corresponding to $\langle t, t' \rangle$ will be

the funct: n which, given u, if there is a substitution σ such that $\sigma(t) = u$, returns $\sigma(t')$, or else fails. These rules will usually be obtained by "lifting" a theorem that is a universally quantified equation.

We first need to develop a theory of substitution for meta-terms. The *variables* of a meta-term t are defined with respect to a list l of identifiers, and are the subterms of t that are applications of a member of l to no arguments. A *substitution* is a member of the type `(Atom # Term0) list`. The application of a substitution *subst* to a meta-term t is written *subst*(t). A matching function is extracted from the proof of the specification

```
∀vars:Atom list. ∀t1,t2:Term0.
?(∃s:Atom#Term0 list where ... & s(t1)=t2 in Term0)
```

(where we have elided an uninteresting conjunct).

The rewrite associated with a "lifted" equation is simply defined in terms of the matching function (call it `match`). For *vars* a list of atoms, and for u, v terms, the function is

```
rewrite{vars}(u->v)  ≡
λt. let s(subst) = match(u,t,vars) in s(subst(v)): ?Term0.
```

(recall that `s(a)` denotes a successful computation). To prove that

```
rewrite{vars}(u->v) in Rewrite(a)
```

for particular u and v, one uses a special tactic that makes use of a collection of related lemmas. This tactic often completes the proof without user assistance.

There are two simple basic rewrites that are used frequently. The first is the trivial rewrite,

```
Id  ≡  λt. s(t).
```

The second is `true_eq`, which rewrites an equality meta-term a=b in A to `True` when a and b are "syntactically" equal.

4.4.3 Building Other Rewriting Functions

The approach to term rewriting of Paulson [13] is easily adapted to our setting. His approach involves using analogues of the LCF *tacticals* [7] as

combinators for building rewrites, where his rewrites produce LCF proofs of equalities. Using these combinators, we can concisely express a variety of rewriting strategies (Paulson used his rewriting package to prove the correctness of a unification algorithm [13]). Furthermore, since we can prove that each combinator is correct (i.e., it combines members `Rewrite`(α) into a member of `Rewrite`(α)), proving that a complex rewriting function is correct often involves only simple typechecking.

If f and g are in `Rewrite`(α), then so are the following.

- f `THEN` g. This rewriting function, when applied to a term t, fails if $f(t)$ fails, otherwise its value is $g(f(t))$.
- f `ORELSE` g. When applied to t, this has value $f(t)$ if $f(t)$ succeeds, otherwise it has value $g(t)$.
- `Progress` f. This fails if $f(t)$ succeeds and is equal (as a member of `Term0`) to t, otherwise it has value $f(t)$.
- `Try` f. If $f(t)$ fails then the value is t, otherwise it is $f(t)$.
- `Sub` f. This applies f to the immediate subterms of t, failing if f failed on any of the subterms.
- `Repeat` f. This is defined below.

In Nuprl, all well-typed functions are total. However, many of the procedures, rewrites in particular, that we will want to write will naturally involve unrestricted recursion, and in such cases we will not be interested in proving termination. For example, many theorem proving procedures perform some kind of search, and searching continues as long as some criteria for progress are satisfied. In such a case, it will often be difficult or impossible to find a tractable condition on the inputs to the procedure that will guarantee termination. It appears that the effort to incorporate partial functions into the Nuprl type theory has been successful [3], but a final set of rules has not yet been formulated and implemented.

For our purposes, there is a simple minded solution to this problem which should work in all cases of practical interest. We simply use integer recursion with a (very) large integer. Thus, for A a type, a a member of

A, and f a function of type A->A, we can define `fix{`A`}(`a`,`f`)` to be an approximation to the fixed-point of f. One difference between this and a true fixed-point operator is that we must take into account the effectively bogus base case a. Using this definition it is easy to define **Repeat**:

```
Repeat(f)  ≡
fix{Term0->?Term0}(Id, λg. (f THEN g) ORELSE Id).
```

We can also directly define, using `Id` as the "base case", a fixed-point operator **rewrite_letrec** for rewriting functions that satisfies

```
∀α:Env. ∀F:Rewrite(α)->Rewrite(α).
      rewrite_letrec(F) in Rewrite(α).
```

As an example of the use of our combinators, we define a rewrite that does a bottom-up rewriting using f:

```
BotUp(f)  ≡  letrec g = (Sub(g) THEN Try(f)).
```

Normalization with respect to f could be accomplished with

```
Repeat (Progress (BotUp(f))).
```

A rewriting function is applied using a special tactic. The subgoals generated by the tactic are the rewritten sequent, and a subgoal to prove that the rewriting function is well-typed. This latter subgoal often requires only simple typechecking that can be handled automatically.

4.5 Other Applications

There are also two smaller implemented applications in our library. First, as an example of a procedure that works on an entire lifted sequent, there is an equality procedure that uses equalities from the lifted hypotheses, and decides whether the equality in the conclusion follows via symmetry, transitivity, and reflexivity. Secondly, there is an implementation of a version of the sorting algorithm described earlier. This algorithm normalizes expressions over a commutative monoid (a set together with a commutative associative binary operation over that set and an identity element). Given

a term of the form $a_1 \cdot a_2 \cdot \ldots a_n$, where $\cdot$ is the binary operation of the monoid, and where no a_i is of the form $b \cdot c$, the algorithm "explodes" the term into the list $[a_1; \ldots; a_n]$, removes any a_i which is the identity element of the monoid, sorts the list, according to a lexicographic ordering on terms induced by an ordering on atoms taken from the *current environment*, and finally "implodes" the list into a right-associated term.

5 Comparison with Other Work

The idea of using a formalized metatheory to provide for user-defined extensions of an inference system has been around for some time. Some of the early work was done by Davis and Schwartz [5] and by Weyrauch [14].

The work that is closest to our own is that of Boyer and Moore on *metafunctions* [2]. Their metafunctions are similar to our rewriting functions. One of the points which most sharply distinguishes our work is that in our case the entire mechanism is constructed within the theory. An important implication is that we can abstract over environments, drastically reducing the burden of proving the correctness of theorem proving procedures. Also, we do not have, as Boyer and Moore do, one fixed way of applying proven facts about the embedded meta-theory. The user is free to extend the mechanisms; for example, to provide for the construction and application of conditional rewrites. Another difference is that the higher-order nature of the Nuprl type theory allows us to construct useful functions such as the rewrite combinators. Finally, the represented class of objects is much richer than what can be represented in the quantifier-free logic of Boyer and Moore.

Constable and Knoblock [11,10] have undertaken an independent program to unify the meta-language and object theory of Nuprl. They have shown how to represent the proof structure and tactic mechanism of Nuprl in Nuprl's type theory. This is largely complementary to the work described here. Given an implementation of their approach, in order to obtain the same functionality provided by our library virtually all of our work would have to be duplicated, although in a more convenient setting. They do not deal with representing contexts, and do not impose the kind of structure on represented terms that allows direct construction of such procedures as

rewriting functions.

6 Conclusion

There are some questions concerning the practicability of this whole reflection-based approach that cannot yet be fully answered. Some small examples have been developed, but conclusive answers will not be possible until the approach is tested in a major application. It is hoped that it will be a useful tool for the implementation of a significant portion of Bishop's book on constructive analysis [1].

One of these questions is whether encodings in Nuprl of reasoning procedures are adequately efficient. The data types these procedures operate over are not significantly different from those used in the implementation of Nuprl, so this question reduces to the open question of whether Nuprl programs in general can be made efficient. In the example given earlier involving rational arithmetic, the actual rewriting (the normalization of the term that is the application of the rewriting function to its argument) took about ten seconds. The normalization was done using Nuprl's evaluator, which employs a naive call-by-need algorithm, and with no optimization of extracted programs.

Another question is whether lifting a sequent and maintaining a lifted sequent are sufficiently easy. Currently, these operations are somewhat slow. For instance, in the example, the lifting step and the rewriting step each took a total of about two to three minutes. The main reason for this problem has to do with certain gross inefficiencies associated with Nuprl's term structure and definition mechanism. Solutions are known and awaiting implementation.

The most difficult and important question is whether it is feasible to formally verify in Nuprl significant procedures for automating reasoning. The term rewriting system gives some positive evidence, as do some of the other examples outlined above that point out the utility of the means for abstraction available in Nuprl.

Acknowledgments

Stuart Allen, David Basin, Bob Constable, and the CADE-9 referee made helpful comments on the presentation of this work. The author is especially grateful to Bob Constable for his support and encouragement.

References

[1] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[2] R. S. Boyer and J Strother Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, chapter 3, Academic Press, 1981.

[3] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, IEEE, 1987.

[4] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] Martin Davis and Jacob T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217-230, 1979.

[6] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration, Lecture Notes in Mathematics vol. 125*, pages 29-61, Springer-Verlag, New York, 1970.

[7] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[8] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*, IEEE, 1987.

[9] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

[10] Todd B. Knoblock. *Metamathematical Extensibility in Type Theory*. PhD thesis, Cornell University, 1987.

[11] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, IEEE, 1986.

[12] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North Holland, Amsterdam, 1982.

[13] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[14] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

Reflecting the Open-Ended Computation System of Constructive Type Theory *

Stuart F. Allen
Robert L. Constable
Douglas J. Howe

Cornell University
Ithaca, NY 14853

Abstract

The computation system of constructive type theory is open-ended so that theorems about computation will hold for a broad class of extensions to the system. We show that despite this openness it is possible to completely reflect the computation system into the language in a clear way by adding simple primitive concepts that anticipate the reflection. This work provides a hook for developing methods to modify the built-in evaluator and to treat the issues of intensionality and computational complexity in programming logics and provides a basis for reflecting the deductive apparatus of type theory.

1 Introduction

In this abstract we use the term "reflection" to refer to constructions which allow a language to talk about its syntax. This capability is important in natural language and is also an important mechanism in formal languages. In Lisp it is used to provide an extensible syntax. Formal logical calculi also use it to provide an extensible inference system [3], by allowing users to state new rules of inference and prove that they are sound. Reflection need not always be explicitly provided for when a language is designed because it can sometimes be achieved through *gödelization*, a technique used by Gödel to prove his incompleteness theorem by reflecting the relation "p is a proof of P" inside the pure language of arithmetic.

Reflection provides not only a basis for reasoning about computation, but also a means of modifying evaluation, say to make it more efficient. For instance it is possible to provide other function evaluation procedures such as "call by value" in addition to the basic lazy evaluation. Reflection also provides a basis for reasoning

*Supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409.

about syntax. We can define basic operators on terms at the reflected level, e.g. substitution, renaming, pattern matching, unification, etc. These can be given just as at the metalevel, providing an internal account of the basic system operators. Also, because the term structure is so general, its reflection provides a way to study syntax and metamathematics rigorously inside the system in such a way that the results are applicable to the reflected system. Moreover, because the theory is constructive, the metatheorems are also applicable.

We believe that reflection will be a good mechanism for treating at least two basic concepts that have proved troublesome in formal programming logics, namely the notion of the *structure* of algorithms, formulas, proofs, etc., and a notion of *computational complexity*. In some ways reflection seems like an obvious way to treat these concepts, but formal reflection mechanisms are subtle and sometimes opaque. So there is a bias away from them (see for example [11,9]), and they have not been used in this way. One of the results of our work is a clean reflection mechanism which will support an account of structural and computational complexity in programming logics.

One reason that we can make the reflection mechanism especially clear is that we describe it in constructive type theory, an exceptionally rich language. One reason it is so useful is that the theory it reflects is so rich, unlike the situation with a quantifier free theory based on Lisp syntax. However, one of the characteristics of this theory is its *open-ended computation* system. The "openness" notion is that the theory should remain sound when new computational forms are added as primitives as long as they satisfy certain simple conditions. This feature of type theory is very important to its role in the foundation of constructive mathematics because in mathematics the notion of construction is never finally closed off. We believe that this feature is also interesting in programming languages and logics. For example, building a programming logic on the principle guarantees that all rules and theorems will remain valid if a new operator is added, for example a new form of iteration, or a non-deterministic choice operator, or a form to represent a more efficient access mechanism to a data structure, or forms to reflect the theory in itself. (For example, the underlying computational system of Nuprl has been extended several times, once to add induction for recursive types [8], again to allow generators for infinite objects [8], then a fixed point operator [10]. Similarly Martin-Löf type theory has been extended twice since 1973 [6], to add well-founded trees [7] and lazy numbers. Because of the open-endedness principle, no previously established theorems became false in these extensions. Other programming languages, notably Lisp, have undergone similar evolution.) Open-endedness will be discussed further in section 3.

In a sense, open-endedness is assumed by programming language designers who freely add new constructs in the evolution of a language, e.g. new procedure calling mechanisms. Difficulties do not show up until one tries to axiomatize the language, then rules must often be amended to take account of new kinds of execution. Constructive type theory is designed from the start to accomodate these extensions without changing the rules already established. Our results show that it is possible to preserve this important design principle and nevertheless allow a reflection mechanism. It is not clear *a priori* that this can be done. This is acheived by providing primitives that

anticipate reflection, as opposed to taking a fixed language and finding a gödelization. we cannot use a fixed encoding of the terms into a fixed class such as the integers because the class of terms can be extended after any such encoding is attempted.

One of the by-products of our analysis is a new method for explaining the syntax of type theory itself. This can be done because the type theoretic notation for inductive definitions and the basic types and sets used to define the theory itself are isomorphic to standard informal accounts of these concepts. So the definition of the syntax and computation rules is presented first in this natural high level notation which then turns out to be exactly the notation being reflected into the internal definition of a formal language. This capability is not so strikingly useful for less rich theories.

We put off the more complex issue of how to reflect the deductive apparatus of type theory, but the practical value of such a reflection is well illustrated by Howe's development of a reflected theory of term rewriting for a fragment of type theory [4,5]. The work described here is a necessary prerequisite to "deductive reflection", and is sufficiently involved that it bears this separate treatment. In particular, this is a good basis for exploring the notion of complete reflection in programming languages.

2 Preliminaries

2.1 Terms

Constructive type theory can be defined starting with a class of terms which will denote the types and their inhabitants. For example, there may be a term for the list induction combinator; following [2] it has the form $list\_ind(l; b; h, t, v.g)$. In this term, l, b and g are *subterms* and h, t, and v are *bound variables* whose scope is g, i.e., all occurrences of these variables in g are *bound*. Typically there will be many term constructors such as pairing of terms a and b, often denoted $< a, b >$, and injecting a term a into a binary dijoint union, often denoted $inl(a)$ or $inr(a)$. In Martin-Löf '82 there are 30 such operators, in Nuprl there are 38. It simplifies matters to regularize term formation. Here we will stipulate that all terms, however they may be written, may be uniquely put into the form $op(\overline{v}_1.t_1; \ldots; \overline{v}_n.t_n)$, where $\overline{v}_i$ is a vector of *bound variables* and t_i is a *subterm*. We consider the *operator*, op, as a component; let us call a term constructed in this way an *instance* of op. If there are n subterms of an instance of an operator, we say that the operator has *arity* n; each operator has a unique arity. And for each argument place i, there is a unique number of binding variables, the length of $\overline{v}_i$, required for binding in to t_i.

Terms are built inductively from variables using operators, constants being instances of operators of arity zero. We treat variables as a special category of term, and indeed, we shall assume they are gotten by injecting into the class of terms, some discrete class of objects, which it will be convenient for us to call *variables* as well. When we must take special care in distinguishing these latter objects we call variables from the terms we call variables, we shall call the terms *variable injections*. The inductive character of terms justifies primitive recursion and proof by induction over the structure of terms.

2.2 Evaluation

Computational meaning in type theory arises from rules for evaluating terms, e.g., saying that $1 + 1$ evaluates to 2. Evaluation is a partial procedure from closed terms to closed terms. A more involved example than $1 + 1$ is needed to see the key points. We consider the evaluation of the list induction form.

If lists are built from nil by the operation of adjoining an element a to a list l, producing a new list $a.l$, then the procedure for evaluating instances of the primitive recursion operator for lists is specified as follows. Let us abbreviate $list\ ind(\ s\ ;\ b\ ;\ h, t, v.g\ )$ by $F(s)$. Then $F(s)$ evaluates to a term e iff either s evaluates to nil and b evaluates to e, or else, for some a and l, s evaluates to $(a.l)$ and $g[a, l, F(l)/h, t, v]$ evaluates to e.

2.3 Open-Endedness

For reasons discussed in the introduction, we are interested in *open-ended computation systems*. This means that we must be able to provide for the introduction of new terms and their incorporation into the evaluation relation. It is not clear how this can be specified in such a way that the system is truly open-ended and yet can be completely described to the point where the mechanism can be reflected into the theory itself. We consider the issue further before introducing our solution.

Let us look more closely at what might be meant by open-ended computation systems. We are aware of two radically different interpretations. One, which we may call *profound* open-endedness, construes open-endedness to be a property of computation systems. On this view, one might also consider other open-ended sorts of systems or classes. One may introduce new elements into an open-ended system (or class) and yet maintain the identity of the system (or class). While we are in no position to promise a satisfactory explanation of open-endedness, we might draw the reader's attention to free-choice sequences as intuitionistic examples of open-ended systems, where one introduces a new branch at will. The need for open-ended computation systems in the *intuitionistic theory of types* (see ML82) arises because the intuitionist does not believe that all methods of computation can be effectively described at once, or at least not now. Hence, the rejection of Church's thesis. Yet, intuitionistic type theory needs a type of all possible representations of, for example, (effectively computable) functions from numbers to numbers. As a result, one must be able to choose new representations of number-theoretic functions and introduce them at will.

The other interpretation of open-endedness of (computation) systems is independent of constructivism. Here, the open-endedness is not a property of individual systems, but rather, consists in pragmatic constraints on our *use* of systems. Whatever theorems and procedures we have developed *within those constraints* for one system, will apply to any *different* system which is a suitable extension of the first. The language of constructive type theory can be given a natural re-interpretation using conventional (non-open-ended) classes, using ZF, say, which preserves the suitability of the language as a programming language and logic [1]. It happens that when one carefully states the method for introducing new terms under the "profound" interpre-

tation, the results are readily re-interpreted as conditions for suitable extension of fixed evaluation systems. Our opinion is that this particular approach even to merely pragmatic open-endedness is quite elegant. Also, the method for reflecting the profoundly open-ended computation system will work just as well for closed systems.

An examination of [7] would reveal that to introduce a new form of expression into the language, one must specify the number of constituent subexpressions required to form expressions of that form, and one must specify which variables become bound in which constituent subexpressions, and one must explain how to evaluate terms of that form. We have refined this procedure, as part of an effort, upon which we hope to report later, to clarify for ourselves a number of issues in the design of type theoretic languages. Here is an approximation adequate for explaining our reflection method. We assume a discrete class of objects we shall call the *variables*. We assume an open-ended class of discrete objects to which we may always, at will, introduce a new member; we shall call the members of this class *operators*. But, whenever one introduces a new operator, one is required to associate with it several other entities that meet several conditions. Introducing a new operator and carrying out these related tasks constitute the procedure for introducing a new form of expression:

1. Introduce a new operator.
2. Indicate a number as the *arity*.
3. Indicate, for each i from 1 to the arity, called the *number of binding variables at place i*.
4. By a *term* we shall mean a (finite) tree each of whose nodes is labelled either with a variable or else with
   - an operator and
   - for each i from 1 to the arity, a vector of variables, the length of which is the number of binding variables at place i.

   Give a partial term-valued procedure on terms; this procedure may apply the evaluation procedure. Further, this procedure must meet several conditions which we will not describe here. We shall call this procedure the *evaluation fragment*.

To evaluate a term t, see if the root is labeled with an operator rather than a variable; if so, then apply the evaluation fragment for that operator to t; if execution results in a term s, then return s.

There is much about this procedure for introducing forms of expression that demands explanation and justification. For example, the availablity of the full evaluation procedure in the evaluation fragments requires a finer analysis. The conditions imposed on the evaluation fragment when introducing it are quite important as well; they are needed to guarantee some global features of the open-ended system, such as idempotence of evaluation, and the harmlessness of changing bound variables. And

of course, the specification of open-ended classes suitable for use as the operator class is not unproblematic.

But, our topic is reflection, and the features of the computation system to be reflected here are adequately represented for the purpose. These are the variables, the open-ended class of operators, the tree structure of terms, and the evaluation procedure. The main thing to notice is that the whole process of computation-system extension is *operator driven*. The open-endedness of the class of terms arises solely from the open-endedness of the class of operators, and the extension of evaluation to new terms is incremental in the operators.

The merely pragmatic open-endedness is realized by interpreting the class of operators in the above procedure as a parameter ranging over fixed discrete classes. To use a fixed computation system in an open-ended fashion, one proves only theorems true for every computation system reachable from it by a series of extensions through the above procedure. (Introducing some new operators would consist of choosing a "new" class of operators and an isomorphism between the members of the "old" class and the "old" members of the "new" class.)

3 Defining Reflection

Our aim is to reflect the tree structure of terms as described above, to reflect the evaluation procedure as a whole, and to reflect the reflection apparatus itself. Before proceeding with that, let us contrast this aim with a variety of other related goals which one might attempt. As was mentioned earlier, we will not carry out the reflection of the deductive apparatus, although it could be done rather directly since the possible formal proofs are inductively definable from the terms.

The more detailed structure of evaluation and evaluation fragments, as well as the constraints on evaluation fragments (which were mentioned but not described), will not be reflected. In particular, although we could reflect the fixed-point construction of evaluation from (further parameterized) evaluation fragments, we found it to be very complex; we do not know of a practical benefit great enough to justify reflecting in such detail. However, we do want to reflect, but not here, a principle of induction over complete evaluations, which is critical to proving important global, i.e., not operator-specific, facts about terms that have values (see [10] for relevant discussion).

We will not try to reflect our knowledge about the reflection mechanism. Although we will be able to use our reflective devices to state propositions about the syntax, even the design criteria to be worked out below for the reflective apparatus itself, we will not concern ourselves here with providing formal deductive apparatus sufficient for proving these propositions (which we know to be true). In practice, it is, of course, of prime importance that sufficient deductive apparatus be designed (by appropriate choice of new axioms or inference rules). We have not yet given much thought to this.

Finally, we do not even partially reflect the semantics of type theory, which must begin with the reflection of type membership.

3.1 Types and Representation

A typical use of types within the language of intutitionistic type theory, indeed, one *might* argue, the only use, is as a collection of expressions referring to constructive objects of a a given sort. For example, one may define a type N for the purpose of collecting represention of some kind for natural numbers, or $N \rightarrow N$ to collect some manner of representations for (effectively computable) numeric functions. We may use a type T to refer not only to itself but also to the sort of objects its members are designed to represent. Two terms t and s which are members of a type T, or $t, s \in T$, are said to be equal in type T, or $t = s \in T$, when they represent the same object of sort T. The intended use of type theoretic language requires that the representation associated with any given type be preserved under evaluation, and that only closed terms can be used as representations.

Here is the representation of natural numbers used for the type N. Zero is represented by any term that evaluates to the term 0. The successor of a number n is represented by any term that evaluates to a term $\mathrm{succ}(t)$, where n is represented by t. This interleaving of evaluation with canonical forms is typical of the method of lazy evaluation. Note that there will be infinitely many representations of each number, and that whether an arbitrary term represents a number is undecidable.

To make use of this representation, the type N is introduced and inhabited by the representations of numbers, equal members of N being terms that represent the same number. We may take the design of the type N as our paradigm for designing types for representing terms.

3.2 Anticipation

We now turn to the question of whether we can completely reflect the open-ended computation system described above. Our hope is that by using type theory (rather than gödelizing) we can offer an especially clear analysis on account of the ease of designing representations and types.

However, looking ahead to designing the representation of terms, we may expect to use a representation for operators, since they are constituents of terms. How can we guarantee that whenever a new operator is introduced we will also have a name for it After considering a few alternatives, we decided to *extend the structure* of terms; for each operator, not only will the instances of that operator be among the terms, but so will a new degenerate form of term gotten simply by injecting the operators into the term class. We shall call such degenerate terms *operator injections*. So, from now on we shall admit three kinds of terms: variable injections, operator injections, and instances of operators. As we shall see, with this single anticipation of the reflection scheme to come, the design will develop quite smoothly.

3.3 Design Criteria

We shall only outline the design and implementation of the reflection scheme in this abstract; it will be more fully presented in the paper. We can determine what new

type and members are needed by describing precisely the computation system as set out in section 2. One can view this effort as an explantion of the basics of computation in the theory itself.

There must be types for *operators*, *variables* and *terms*. These will be denoted $Op, Var, Term$. (In general we notate the internal reflective notions by capitalizing the first letter of the informal type used in the metalanguage.) Var should be a discrete type handy enough to use in typical syntactic operations on variables, such as finding an unused one. The handiest would be lists over some finite alphabet, perhaps *LetterList*, but for purpose of demonstration, we shall not require this here.

$Term$ can be easily defined as a recursive type given the types Var and Op and also representations of the arity and binding lengths for each operator:

Arity: $Op \rightarrow N$,

Bvnum: $o : Op \rightarrow i : [1, Arity(o)] \rightarrow N$

(Here we employ the dependent function space constructor to give an exact typing of these functions.)

So, defining $Term$ boils down to defining Op; but this is just what we anticipated. Define Op by stipulating that $a = b \in Op$ iff, for some operator op, a and b evaluate to the operator injection of op.

The evaluator is represented as a partial function by two components. There is a relation, i.e., a 2-place type-valued function, to represent evaluation, and a function, Val, to compute the value of a term when it exists. The type s *Evaluates_to* t should be inhabited when s represents a term s', and t represents a term t', and s' evaluates to t'.

The term $Val(s)$ is to be defined as follows, assuming that we have a procedure for choosing a *standard* representative of a term: to evaluate $Val(s)$, see if s represents a terms s'; if so, then see if s' evaluates to some t'; if so, then return the standard representative of term t'. Thus, the implementation of Val only awaits the design and implementation of $Term$ (term representation) and a standard-representation function.

All that remains to design is the reflection of the rest of the reflection apparatus, namely, term representation and standard term-representation. The specification for s *Reps* t is analogous to that for s *Evaluates_to* t, and $Rep(s)$ should be designed to represent the standard representation function, much like $Val()$ represents the evaluation function (only, $Rep(s)$ is total on terms).

3.4 Implementing Reflection

Again, in this abstract we shall be very brief in discussing the implementation that will appear in the paper. We want to specify a particular way to meet the design criteria listed above. We have a particular sort of type theory in mind. It should be an open-ended extensional system with at least the primitives of Martin-Löf '82 or Nuprl. Depending on certain other details, there are various ways to proceed. For example, if there are recursive types, then $Term$ can be defined naturally using them, otherwise it must be encoded somewhat, say using the W-types of Martin-Löf '82, or

even using quotient or set types of [2], along with the other usual constructors. If there are partial types, as in extensions of Nuprl [2,10], then Val can be interpreted as a partial function. Otherwise we can use *Evaluates_to* to specify its domain exactly and make it a total function.

We shall suppose our formal variables to be simply natural numbers, and we therefore implement Var as N.

There is no significant latitude for implementing $s\ Evaluates\_to\ t$ and $Val(s)$. They are simply introduced as primitives in the most direct possible way.

It turns out that $s\ Rep\ t$ and $Rep(t)$ need not be introduced as primitive since given the other primitives, they are expressible in terms of the other conventional operators by recursion on $Term$.

To sum up, the only primitives we introduce are the type Op, exploiting operator injection, the operator $Val(t)$, and the type constructor $s\ Evaluates\ t$. From these, together with conventional operators and type constructors, all the rest of our reflection apparatus can be expressed. Although in the final paper we will show that the implementation meets the design and that the reflection mechanism itself can be reflected, we feel that it is the design, described in some detail above, which conveys most of the novel ideas.

References

[1] S. F. Allen. *A non-type-theoretic semantics for type-theoretic language*. PhD thesis, Cornell University, 1987.

[2] R. L. Constable, S. Allen, H. Bromley, W. Cleveland, and et al. *Implementing Mathematics with the Nuprl Development System*. NJ:Prentice Hall, 1986.

[3] M. Davis and J. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. In *Comp. Math. with Applications, v. 5*, pages 217–230, 1979.

[4] D. Howe. Computational metatheory in Nuprl. *CADE-9*, 230:404–415, 1988.

[5] D. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.

[6] P. Martin-Lof. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73.*, pages 73–118. Amsterdam:North-Holland, 1973.

[7] P. Martin-Lof. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. Amsterdam:North Holland, 1982.

[8] P. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[9] N. Shanker. Towrds mechanical metamathematics. Technical report, Institute for Computing Science, University of Texas at Austin, 1984. Tech. report 43.

[10] S. Smith. *Partial Objects in Type Theory.* PhD thesis, Cornell University, 1989.

[11] R. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.